НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

КАФЕДРА СИСТЕМНОГО ПРОГРАМУВАННЯ І
СПЕЦІАЛІЗОВАНИХ КОМП'ЮТЕРНИХ СИСТЕМ

«На правах рукопису»
УДК _____

«До захисту допущено»

Завідувач кафедри СПСКС

_____ В.П.Тарасенко_
(підпис) (ініціали, прізвище)

"___"_____2018р.

Магістерська дисертація

на здобуття ступеня магістра

зі спеціальності 123 Комп'ютерна інженерія
(Системне програмування)

на тему: Методи динамічного завантаження класів у розподілених сховищах даних

_____
_____
_____

Виконав (-ла): студент (-ка) ІІ курсу, групи _КВ-62м_____
(шифр групи)

Ріпневський Олександр Олекандрович ____ _____
(прізвище, ім'я, по батькові) (підпис)

Науковий керівник Доцент кафедри СПіСКС, к.т.н., доц. Замятін Д.С. _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Рецензент_____ _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2018 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет прикладної математики

Кафедра системного програмування і спеціалізованих комп'ютерних систем

Рівень вищої освіти – другий (магістерський)
Спеціальність 123 Комп'ютерна інженерія
(Системне програмування)

ЗАТВЕРДЖУЮ
Завідувач кафедри СПСКС
_____ _В.П.Тарасенко__
(підпис) (ініціали, прізвище)
«___»_____2018р.

ЗАВДАННЯ
на магістерську дисертацію студенту
_____ Ріпневський Олександр Олександрович_____
(прізвище, ім'я, по батькові)

1. Тема дисертації Методи динамічного завантаження класів у розподілених сховищах даних_____
_____ ,
науковий керівник дисертації Замятін Денис Станіславович, к.т.н, доцент ,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від «22» березня 2018 р. №986-с

2. Термін подання студентом дисертації 11 травня 2018 р.
_____

3. Об'єкт дослідження технологія динамічного завантаження класів у розподілених сховищах даних
_____

4. Предмет дослідження In-Memory Data Grid реалізації розподілених сховищ даних
_____

5. Перелік завдань, які потрібно розробити аналіз існуючих реалізацій, розробка алгоритму та моделі,створення програмного модуля
_____

6. Перелік ілюстративного матеріалу діаграма класів , діаграма класів базової моделі, алгоритм роботи базової моделі, алгоритм роботи моделі , алгоритм роботи програми, діаграма залежності модулей

7. Перелік публікацій Збірник матер. Всеукр. наук.-практ. WEB-форуму «Проблема зміни схеми даних для систем з In-Memory Data Grid архітектурою», ICSFTI2018 «Використання СУБД із архітектурою In-Memory Data Grid»

8. Дата видачі завдання 5 вересня 2016 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
	Вивчення літератури за тематикою проекту	6.09.2016	
	Розроблення та узгодження технічного завдання	12.12.2016	
	Аналіз існуючих рішень	1.04.2017	
	Підготовка матеріалів першого розділу магістерської дисертації	23.04.2017	
	Підготовка матеріалів другого розділу магістерської дисертації	20.08.2017	
	Підготовка матеріалів третьго розділу магістерської дисертації	21.10.2017	
	Підготовка матеріалів четвертого розділу дипломного проекту	20.01.2018	
	Підготовка графічної частини дипломного магістерської дисертації	20.02.2018	
	Оформлення документації магістерської дисертації	23.03.2018	
	Попередній розгляд магістерської дисертації на кафедрі	26.04.2018	

Студент _____ Ріпневський О.О.
(підпис) (ініціали, прізвище)

Науковий керівник дисертації _____ Замятін Д.С.
(підпис) (ініціали, прізвище)

РЕФЕРАТ

Актуальність теми.

Сучасне суспільство дедалі більше набирає рис інформаційного. Інформація стає одним із головних стратегічних ресурсів держав на одному рівні з матеріальними, енергетичними та електронними.

Обробка даних безпосередньо в оперативній пам’яті є досить широко обговорюваною темою в останній час. Багато компаній, які в минулому відмовлялись розглядати використання in-memory технологій через високу вартість, зараз перебудовують архітектуру своїх інформаційних систем, щоб використовувати переваги швидкої транзакційної обробки даних, пропонованих даними рішеннями. Це є наслідком стрімкого падіння вартості оперативної пам'яті (RAM), в результаті чого стає можливим зберігання всього набору операційних даних в пам'яті, збільшуючи швидкість їх обробки більш ніж в 1000 разів, порівняно із обробкою даних на жорстких дисках. In-Memory Compute Grid та In-Memory Data Grid продукти надають необхідні інструменти для побудови таких рішень.

Цей підхід дуже швидко набув широкого визнання серед експертів в області проектування хмарних платформ, а також будь-яких систем, що мають потребу в практично необмеженому масштабуванні системи зберігання даних. Багато відомих компаній випустили на ринок системи такого типу:

Oracle Coherence - Java / C / .NET

VMWare Gemfire - Java

GigaSpaces - Java / C / .NET

JBoss (RedHat) Infinispan - Java

Terracota - Java

В межах даного дослідження будуть розглядатись рішення для Java, тобто вузлами кластера IMDG будуть JVM. Крім того IMDG на Java може бути використаний для швидкого доступу до даних через REST API.

В даній роботі розглянуті різні реалізації розподілених сховищ даних. Описано основні функції та розглянуто механізми обробки даних в IMDG. Окремо розглянутий механізм завантаження класів до сховищ в оперативній пам’яті, та описано існуючі рішення його оптимізації.

Технологія та In-Memory Data Grid є відносно новою, тому існує ряд проблем, які потребують вирішення. Однією з таких проблем розподілених сховищ даних в оперативній пам’яті є неможливість зміни схеми даних без перезавантаження кластеру, що, в свою чергу, призведе до втрати даних. На сьогодні ця проблема вирішується за допомогою динамічного завантаження класів у розподілених сховищах даних. Існує досить велика кількість різнопланових задач, яка вимагає створення системи динамічного завантаження класів у розподілених сховищах даних в оперативній пам’яті.

Об’єктом дослідження є технологія динамічного завантаження класів у розподілених сховищах даних.

Предметом дослідження є In-Memory Data Grid реалізації розподілених сховищ даних

Мета і задачі дослідження. Метою даної роботи є оптимізація механізму та створення системи завантаження класів до розподілених сховищ даних в оперативній пам’яті. Для цього розробляється алгоритм та модель інструменту динамічного завантаження, спрямована на покращення та збільшення швидкості розробки за допомогою IMDG технології.

Для досягнення поставленої мети були поставлені та вирішені наступні задачі:

1. - Проаналізувати технічну та наукову літератури із тематики дослідження.
2. Побудувати теоретичну модель алгоритму роботи інструменту динамічного завантаження та розглянути особливості її застосування.

3. Розробити алгоритм роботи інструменту динамічного завантаження класів без втрати даних та визначити його ефективність.
4. Створити програмний модуль реалізації інструментарію динамічного завантаження класів у розподілених сховищах даних.

Методи досліджень. Для реалізації рішення було створено тестову програму на мові програмування Java та з використанням фреймворку Spring.В якості тестових даних використовувались реальні дані, що були змодельовані на тестову машину.

Наукова новизна одержаних результатів полягає в наступному:

– Запропоновано метод вирішення проблеми динамічного завантаження;

– Проведено тестування на робочих машинах, що наразі використовуються у обчислювані на робочих даних, при цьому було показано деяке покращення продуктивності.

Практична цінність одержаних результатів полягає в наступному:

– Запропонована реалізація вирішує проблему використання машиного часу обробки інформації за рахунок ефективного керування виконавчими машинами;

- на основі результатів можемо стверджувати ефективність покращена в деяких випадках в 2,5 разів.

Структура і обсяг роботи. Робота складається зі вступу, чотирьох розділів, та загальних висновків по роботі, списку використаної літератури з 30 найменувань та чотирьох додатків.

Повний обсяг магістерської дисертації – 105 сторінок, в тому числі 95 сторінок пояснювальної записки, 10 рисунків і 5 таблиць.

У вступі подано загальну характеристику роботи, зроблено оцінку сучасного стану проблеми, обґрунтовано актуальність напрямку дослідження,

сформульовано мету і задачі дослідження, показано наукову новизну і практичну цінність отриманих результатів.

В першому розділі наведений огляд існуючих джерел та методів реалізації.

В другому розділі була розроблена модель системи, що базується на алгоритмі динамічного завантаження.

В третьому розділі була реалізована модель інструмету динамічного завантаження.

В четвертому розділі була впроваджено систему динамічного завантаження.

У висновках наведено узагальнений аналіз результатів, описаних у третьому розділі; визначені переваги і недоліки запропонованого способу.

Додатки містять копії графічних матеріалів, копії публікацій, фрагменти лістингу програми та довідку про впровадження.

Ключові слова: динамічне завантаження, обробка інформації.

Зміст

Список термінів і скорочень

ACID	набір властивостей, що гарантують надійну роботу транзакцій бази даних: атомарність, узгодженість, ізольованість, довговічність. В контексті баз даних, послідовність операцій з базою даних, яка задовольняє властивостям ACID, можна розглядати як одну логічну операцію над датами. Така послідовність операцій називається транзакцією. Наприклад, переказ коштів з одного банківського рахунку на інший, містить численні операції, але є єдиною транзакцією.
IMDG	In-memory data grid - розподілені сховища даних в оперативній пам'яті
InfiniBand	високошвидкісна комутована послідовна шина, що застосовується як для внутрішніх (внутрішньосистемних), так і для міжсистемних з'єднань. Описи InfiniBand специфіковані, підтримкою і розвитком специфікацій займається InfiniBand Trade Association
JVM	Віртуальна машина Java (англ. Java Virtual Machine; JVM) — набір комп'ютерних програм та структур даних, що використовують модель віртуальної машини для виконання інших комп'ютерних програм чи скриптів.
NoSQL підхід	База данних, дані в якій пов’язані не реляційними відносинами (доступ по ключу, графи, документи тощо)
OQL	object query language – об'єктно орієнтована мова запитів (об'єктне розширення мови SQL, що дозволяє будувати запити не тільки до реляційної, але і до об'єктної бази даних)
RAM	Оперативна пам'ять – швидкодіюча пам'ять, призначена для запису, зберігання та читання інформації у процесі її обробки.
REST API	REST (скор. англ. Representational State Transfer, «передача репрезентативного стану») – підхід до архітектури мережевих протоколів, які забезпечують доступ до інформаційних ресурсів.
Сховище даних	Предметно орієнтований, інтегрований, незмінний набір даних, що підтримує хронологію і здатний бути комплексним джерелом достовірної інформації для оперативного аналізу та прийняття рішень. В основі концепції сховища даних (СД) лежить розподіл інформації, що використовують в системах оперативної обробки даних (OLTP) і в системах підтримки прийняття рішень. Такий розподіл дозволяє оптимізувати як структури даних оперативного зберігання для виконання операцій введення, модифікації, знищення та пошуку, так і

	структури даних, що використовуються для аналізу. Уільман Г. Інмон детально описав дану концепцію в своїй монографії «Побудова сховищ даних» в 1992 році.
ЦП	Центральний процесор, ЦП (англ. Central processing unit, CPU) – функціональна частина комп'ютера, що призначена для інтерпретації команд.

Вступ

Актуальність дослідження. Сучасне суспільство дедалі більше набирає рис інформаційного. Інформація стає одним із головних стратегічних ресурсів держав на одному рівні з матеріальними, енергетичними та електронними.

Обробка даних безпосередньо в оперативній пам’яті є досить широко обговорюваною темою в останній час. Багато компаній, які в минулому відмовлялись розглядати використання in-memory технологій через високу вартість, зараз перебудовують архітектуру своїх інформаційних систем, щоб використовувати переваги швидкої транзакційної обробки даних, пропонованих даними рішеннями. Це є наслідком стрімкого падіння вартості оперативної пам'яті (RAM), в результаті чого стає можливим зберігання всього набору операційних даних в пам'яті, збільшуючи швидкість їх обробки більш ніж в 1000 разів, порівняно із обробкою даних на жорстких дисках. In-Memory Compute Grid та In-Memory Data Grid продукти надають необхідні інструменти для побудови таких рішень.

Цей підхід дуже швидко набув широкого визнання серед експертів в області проектування хмарних платформ, а також будь-яких систем, що мають потребу в практично необмеженому масштабуванні системи зберігання даних. Багато відомих компаній випустили на ринок системи такого типу:

Oracle Coherence - Java / C / .NET

VMWare Gemfire - Java

GigaSpaces - Java / C / .NET

JBoss (RedHat) Infinispan - Java

Terracota - Java

В межах даного дослідження будуть розглядатись рішення для Java, тобто вузлами кластера IMDG будуть JVM. Крім того IMDG на Java може бути використаний для швидкого доступу до даних через REST API.

В даній роботі розглянуті різні реалізації розподілених сховищ даних. Описано основні функції та розглянуто механізми обробки даних в IMDG. Окремо розглянутий механізм завантаження класів до сховищ в оперативній пам’яті, та описано існуючі рішення його оптимізації.

Технологія та In-Memory Data Grid є відносно новою, тому існує ряд проблем, які потребують вирішення. Однією з таких проблем розподілених сховищ даних в оперативній пам’яті є неможливість зміни схеми даних без перезавантаження кластеру, що, в свою чергу, призведе до втрати даних. На сьогодні ця проблема вирішується за допомогою динамічного завантаження класів у розподілених сховищах даних. Існує досить велика кількість різнопланових задач, яка вимагає створення системи динамічного завантаження класів у розподілених сховищах даних в оперативній пам’яті.

Метою цієї роботи є оптимізація механізму та створення системи завантаження класів до розподілених сховищ даних в оперативній пам’яті. Для цього розробляється алгоритм та модель інструменту динамічного завантаження, спрямована на покращення та збільшення швидкості розробки за допомогою IMDG технології.

Предметом дослідження є In-Memory Data Grid реалізації розподілених сховищ даних, а **об’єктом** – технологія динамічного завантаження класів у розподілених сховищах даних.

Виходячи з мети роботи, були визначені наступні завдання:

1. Проаналізувати технічну та наукову літератури із тематики дослідження.
2. Побудувати теоретичну модель алгоритму роботи інструменту динамічного завантаження та розглянути особливості її застосування.
3. Розробити алгоритм роботи інструменту динамічного завантаження класів без втрати даних та визначити його ефективність.

4. Створити програмний модуль реалізації інструментарію динамічного завантаження класів у розподілених сховищах даних.

Практичне значення дослідження полягає в тому, що

1. розроблено та впроваджено модель динамічного завантаження класів у розподілених сховищах даних;
2. створено програмний додаток на мові java, що реалізує динамічне завантаження класів у розподіленому сховищі даних Apache Geode.

Структура роботи. Дипломна робота складається зі вступу, трьох розділів, висновків, списку використаних джерел (____ найменувань). Основний текст дослідження викладено на 87 сторінках. Робота містить ___ рисунків.

Розділ 1. Огляд наукових джерел з проблематики динамічного завантаження класів у розподілених сховищах даних

1.1. Теоретичний аналіз In-Memory Data Grid (IMDG) реалізації розподілених сховищ даних

Системи управління базами даних в пам'яті (СУБД в пам'яті, англ. - in-memory database systems, IMDS) це величезний сегмент світового ринку СУБД. Створення СУБД безпосередньо в оперативній пам'яті стало відповіддю на виникнення нових завдань, які стоять перед додатками, новими системними вимогами та операційним середовищем.

СУБД у пам'яті це система управління базами даних, яка зберігає інформацію безпосередньо в оперативній пам'яті. Це радикально контрастує з підходом традиційних СУБД, які розроблені для зберігання даних на стабільних носіях. Оскільки процеси обробки даних у оперативній пам'яті протікають швидше, ніж звернення до файлової системи та зчитування інформації з неї, СУБД в пам'яті забезпечує на порядок більш високу продуктивність програмних додатків. Оскільки конструкція СУБД в пам'яті набагато простіша традиційних, то вони накладають набагато менші вимоги до об'єму пам'яті та характеристик ЦП.

Традиційні СУБД для прискорення роботи широко використовують кешування. Кешування це процес, в рамках якого традиційні СУБД зберігають часто використовувані записи в пам'яті для швидкого доступу до них. Однак, кешування прискорює лише процес пошуку необхідної інформації, а не її обробки. Відповідно, виграш у продуктивності істотно менший. Крім того, керування кеш-пам'яттю само по собі є ресурсоємним процесом, що задіює значні обсяги пам'яті та обчислювальних потужності процесора.

В якості тимчасового рішення розміщення всієї бази даних на диску RAM може прискорити запис і читання бази. Проте, у такого підходу є ряд недоліків.

Зокрема, база даних все ще буде прив'язана до дискового накопичувача, і процеси в базі даних, такі як кешування та введення / виведення даних, будуть здійснюватися, навіть якщо вони вичерпані. Крім того, до бази даних, розміщеної на диску, адресується безліч запитів, вони вимагають часу і ресурсів процесора, і цього не можна уникнути у випадку з традиційним СУБД, навіть якщо вони розміщені в ОЗУ. Навпаки, СУБД в пам'яті використовують єдиний дата-трансфер, що спрощує обробку даних. Видалення зайвих копій даних знижує навантаження на пам'ять, а також підвищує надійність процесу і мінімізує вимоги до ЦП.

Саме тому, в останнє десятиліття, у зв'язку з постійно зростаючими вимогами до швидкості доступу до даних, широкого поширення набули розподілені сховища даних в оперативній пам'яті (in-memory data grid,IMDG). Первісна ідеологія таких сховищ передбачала вибірку даних з IMDG виключно методом доступу по ключу (т.зв. NoSQL підхід). Також була передбачена можливість виконання призначених для користувача функцій безпосередньо на серверах сховища. Однак досвід застосування таких сховищ показав необхідність виконання SQL-подібних запитів. На поточний момент в найбільш поширених реалізаціях IMDG (VmWare Gemfire, Oracle Coherence, Hazelcast) частково підтримується об'єктна мова запитів (object query language, OQL), але жодна з реалізацій розподілених сховищ в оперативній пам'яті не передбачає можливості виконання злиттів розподілених таблиць. Операція join (об'єднання) можна лише за допомогою таблиць, копії яких зберігаються на вузлах сховища цілком (реплікованих таблиці) [1]. Таким чином, в разі необхідності виконання об'єднання розподілених таблиць, користувач змушений виконувати цю операцію самостійно.

Сучасні технології передачі даних можуть бути повільними в порівнянні з обробкою даних в оперативній пам'яті. Мережа, побудована за технологією InfiniBand з номінальною пропускною спроможністю 40 Гбіт / сек, показала 3

Гбіт / сек на вузлі під час виконання секціонування даних за допомогою хешфункції. При виконанні в оперативній пам'яті, секціонування на кілька тисяч частин виконується зі швидкістю, приблизно рівної пропускної здатності оперативної пам'яті [2,3]. Отже, алгоритм об'єднання таблиць для сховищ в оперативній пам'яті повинен мінімізувати пересилання даних по мережі.

Останнім часом інтерес до хмарних архітектур зростає з кожним днем, тому що це один з найбільш ефективних способів масштабування додатків, не докладаючи великих зусиль, а найвужчим місцем будь-якого високонавантаженого проекту є сховище даних, зокрема реляційна БД. Для боротьби з недоліками традиційних БД в основному використовується 2 підходи:

1) Кешування результатів виконання запитів

переваги: висока швидкість доступу до даних

недоліки: вимагає компромісу між актуальністю даних і швидкістю доступу, тому що дані в кеші можуть втратити свою актуальність, а видаляти старі дані з кешу з подальшим кешуванням нових - це додаткові затримки і навантаження на систему

2) NoSQL рішення

переваги: хороша горизонтальна масштабованість, доменна модель даних збігається з моделлю зберігання даних

недоліки: низька швидкість отримання результатів в разі використання диска, практично неможливо забезпечити роботу внутрішньокорпоративного софту, який орієнтований на роботу з конкретною реляційної БД.

Довгий час поняття «грід» означало або добровільне об'єднання обчислювальних ресурсів, або середовища виконання наукових розрахунків, але сьогодні стало ясно, що об'єднання в єдину розподілену інфраструктуру ресурсів пам'яті серверів, що входять в кластер, дозволяє прискорити роботу в режимі реального часу зі структурами пам'яті великих обсягів.

Завдання In-Memory Data Grid (IMDG) - забезпечити надвисоку доступність даних за допомогою зберігання їх в оперативній пам'яті в розподіленому стані. Сучасні IMDG здатні задовольнити більшість вимог до обробки великих масивів даних.

Спрощено, IMDG - це розподілене сховище об'єктів, схоже по інтерфейсу зі звичайною багатопотокової хеш-таблицею. Ви зберігаєте об'єкти по ключам. Але, на відміну від традиційних систем, в яких ключі і значення обмежені типами даних «масив байт» і «рядок», в IMDG Ви можете використовувати будь-який об'єкт з Вашої бізнес-моделі в якості ключа або значення. Це значно підвищує гнучкість, дозволяючи Вам зберігати в Data Grid в точності той об'єкт, з яким працює Ваша бізнес-логіка, без додаткової серіалізації / десеріалізації, яку вимагають альтернативні технології. Це також спрощує використання Вашого Data Grid-a, оскільки в більшості випадків Ви можете працювати з розподіленим сховищем даних як зі звичайною хеш-таблицею. Можливість працювати з об'єктами з бізнес-моделі безпосередньо - одне з основних відмінностей IMDG від In-Memory баз даних (IMDB). В останньому випадку користувачі все ще змушені здійснювати об'єктно-реляційне відображення (Object-To-Relational Mapping), яке, як правило, призводить до значного зниження продуктивності.

Є й інші функціональні особливості, які відрізняють IMDG від інших продуктів, таких як IMDB, NoSql або NewSql бази даних. Одна з основних - по-справжньому масштабується секціонування даних (Data Partitioning) в кластері. IMDG по суті являє собою розподілену хеш-таблицю, де кожен ключ зберігається на строго певному сервері в кластері. Чим більше кластер, тим більше даних можна в ньому зберігати. Принципово важливим у цій архітектурі є те, що обробку даних слід проводити на тому ж сервері, де вони розташовані (локально), виключаючи (або зводячи до мінімуму) їх переміщення по кластеру. Фактично, при використанні добре спроектованого IMDG, переміщення даних буде повністю відсутні за винятком випадків, коли в кластер додаються нові

сервера або видаляються існуючі, змінюючи тим самим топологію кластера і розподіл даних в ньому.

Вперше розробки in-memory СУБД були розпочаті в 1993 році в Bell Labs. Система була прототипована як Dali Main-Memory Storage Manager. Ці дослідження поклали початок створенню першої комерційної in-memory СУБД - Datablitz.

У наступні роки in-memory СУБД привернули увагу найбільших гравців ринку баз даних. TimesTen, компанія-стартап, заснована Марі-Енн Неймат (Marie-Anne Neimat) в 1996 році що відгалуження від Hewlett-Packard, була придбана Oracle в 2005 році. Сьогодні Oracle продає цей продукт у тому числі як in-memory СУБД. У 2008 році IBM купила SolidDB in 2008, також веде роботу в області in-memory СУБД і Microsoft.

VoltDB, заснована одним з піонерів ринку СУБД Майклом Стоунбрейкер (Michael Stonebraker), анонсувала вихід in-memory СУБД в травні 2010 року, на даний момент компанія пропонує як вільну, так і пропрієтарних версію цієї системи. SAP випустила in-memory СУБД, SAP HANA, в червні 2011 року.

В наш час розподілені сховища даних в оперативній пам’яті (In-memory data grid, IMDG) стають все більш поширеними. Основним завданням цієї технології є зменшення часу доступу до даних, завдяки їх збереженню на оперативному запам’ятовуючому пристрої (ОЗП) в розподіленому стані [2].

В межах даної роботи, під IMDG будемо розуміти кластерне key-value сховище, де всі об’єкти зберігаються за ключами, але на відміну від традиційних систем, де ключі значень обмежені типами даних “масив байт” та «строка», в IMDG можливе використання будь-якого об’єкту бізнес-моделі в якості ключа або значення. Це суттєво підвищує гнучкість та дозволяє зберігати в Data Grid саме той об’єкт, з яким працює бізнес логіка, без додаткової серіалізації, якої потребують альтернативні технології. Це також спрощує використання Data

Grid, оскільки в більшості випадків завдяки цьому можна працювати з розподіленим сховищем даних як зі звичайною хеш-таблицею [4].

Ще однією функціональною особливістю IMDG від інших подібних технологій є масштабоване секціонування даних (Data Partitioning) в кластері, що проілюстровано на рисунку 1.1. IMDG по суті представляє собою розподілену хеш-таблицю, де кожен ключ зберігається виключно на конкретному сервері у кластері. Чим більший кластер – тим більше даних можливо зберігати у ньому [6]. Принципово важливим у такій архітектурі є те, що обробку даних необхідно виконувати на тому ж сервері, де вони знаходяться (локально), виключаючи їх переміщення по кластеру. Отже, при використанні добре спроектованого IMDG, переміщення даних буде повністю відсутнім, за винятком ситуацій, коли в кластер додаються нові сервери або видаляються існуючі, змінюючи тим самим топологію кластера і розподілення даних у ньому.

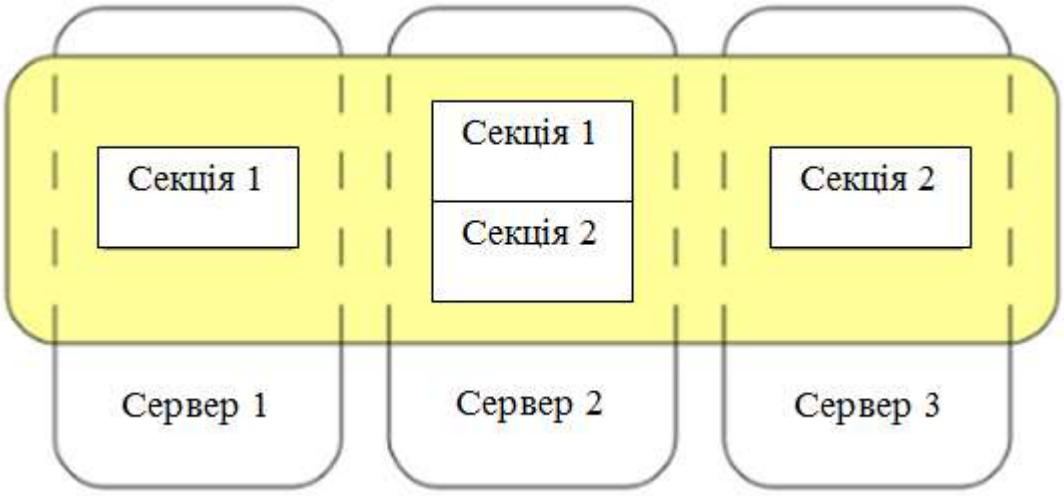

Рисунок 1.1 – Секціонування в розподілених сховищах даних

Основною структурною одиницею IMDG є кеш або так званий регіон, рисунок 1.2. Кеш розподіленого сховища даних в оперативній пам’яті – це розподілений асоціативний масив. Цей масив, на відміну від суворо типізованих реляційних баз даних, зберігає серіалізовані об’єкти, що дозволяє виключити витрати на десеріалізацію з боку клієнта при зчитуванні даних зі сховища. Така

організація дозволяє забезпечити високий рівень горизонтальної масштабованості, що особливо важливо для новітніх веб-сервісів [2].

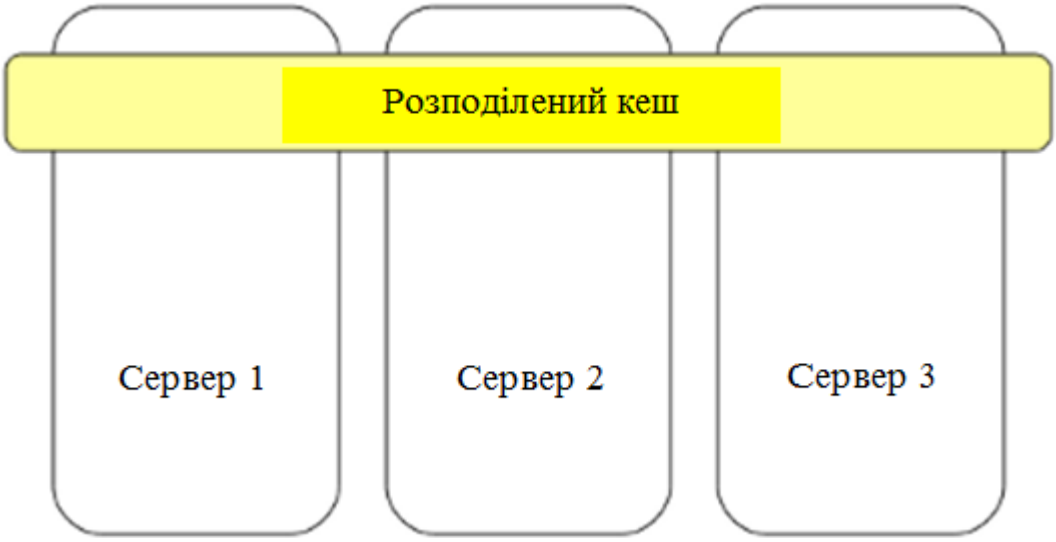

Рисунок 1.2 – Організація кешу в розподілених сховищах даних

Основною парадигмою для IMDG систем є BASE – basically available, soft state, eventual consistency. Ця парадигма була розроблена спеціально для горизонтально масштабованих розподілених систем. Якщо говорити про узгодженість, то згідно цієї парадигми, консистентність даних знаходиться у стані потоку, відповідно постійно змінюється. Сховище зобов'язане забезпечити лише кінцеву узгодженість даних, таким чином дані будуть узгоджені при умові відсутності змін навіть через певний час після останніх змін. Завдяки цьому, у таких системах дані не завжди будуть доступні. Коли дані знаходяться в різних частинах сховища і не прийшли до стану узгодженості, запити до них можуть не повертати результатів [1].

Окремо слід виділити підтримку транзакційності IMDG, що задовольняє парадигми ACID. Як правило, для саме забезпечення механізму узгодженості даних в кластері, використовують двофазну фіксацію. Різні IMDG можуть мати різні механізми блокування, але найбільш новітні реалізації зазвичай використовують паралельні блокування, тим самим зводячи мережевий обмін до

мінімуму та гарантуючи транзакційну узгодженість ACID зі збереженням високої продуктивності.

Різні реалізації технології IMDG мають багато спільних базових функціональних можливостей, але також існує величезна кількість додаткових можливостей, що залежать від конкретного продукту. Особливу увагу слід звернути на механізми вигрузки даних при переповненні (eviction policies), технології завантаження даних, у тому числі при старті сервера ((pre)loading techniques), паралельне розподілення на секції (concurrent repartitioning), об'єм додаткової пам'яті, що необхідний для збереження записів (data overhead).

Окремо слід винести можливість виконання запитів (query) до кешу під час виконання. Наприклад, існують системи, що дозволяють виконувати запити, використовуючи звичайний SQL з підтримкою розподілених join-ов (distributed joins), що на даний момент є рідкістю.

Також частина IMDG дає можливість підтримки об'єктної мови запитів (OQL), про те функціональне наповнення та синтаксис цієї мови відрізняється від однієї реалізації до іншої. Таким чином, при зміні однієї реалізації такої системи на іншу необхідно частково або повністю змінювати використовувані запити, що буває досить проблематично з практичної точки зору. Також через нестандартизовність мови запитів доволі проблематично порівнювати продуктивність різних IMDG систем між собою [2].

Зберігання даних в IMDG – це лише половина функціоналу, необхідного для in-memory архітектури. Дані, що зберігаються в IMDG, необхідно обробляти паралельно та з високою швидкістю. Зазвичай in-memory архітектура секціонує дані в кластері за допомогою IMDG, а згодом виконуваний код відправляється саме на ті сервера, де знаходяться необхідні йому дані. Оскільки виконуваний код (обчислювальна задача) зазвичай є частиною обчислювальних кластерів (Computer Grids), і повинен бути правильно розвернутий (deployment), бути збалансованим при нагрузці (load-balancing), бути відмовостійким (fail-over), а

також мати можливість запуску по розкладку (scheduling), інтеграція між Compute Grid та IMDG дуже важлива. Найбільш ефективною є система, в якій IMDG та Compute Grid є частинами одного і того же продукту та використовують одні, й ті ж сами API, це дозволяє добитися найбільшої продуктивності та надійності in-memory рішення.

1.2. Сучасний стан використання In-Memory Data Grid (IMDG) , як технології системы управления базами данных in-memory

Відповідно до опублікованих тестів McObject, в ході якого порівнювали продуктивність одного і того ж додатку, перенесення традиційної СУБД в RAM дозволило прискорити зчитування даних в 4 рази та оновлення бази в 3 рази в порівнянні з традиційним СУБД на жорсткому диску. У пам'яті СУБД показала ще більш істотні результати порівняння з СУБД на RAM дисків: читання баз даних було в 4 рази швидше, а запис в базу даних виявився швидше в 420 разів.

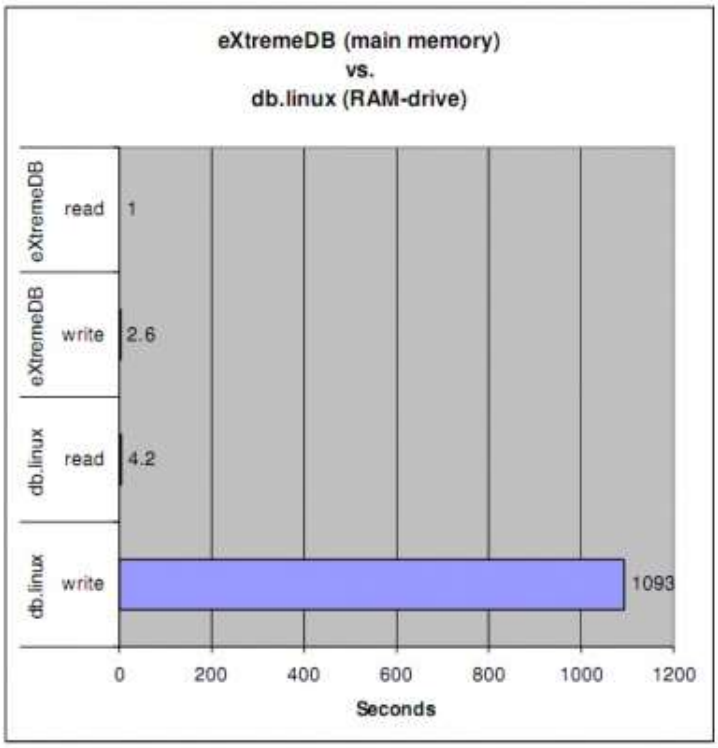

Рисунок 1.3 – продуктивність в пам'яті СУБД eXtremeDB в порівнянні з СУБД db.linux на RAM диску.

Основною відмінністю in-memory СУБД від традиційних СУБД є те, що In-memory СУБД не несе на собі ніякого навантаження від операцій введення / виводу даних. Тому архітектура таких баз даних більш раціональна, а процеси завантаження пам'яті і цикли процесора оптимізовані.

In-memory СУБД зазвичай використовуються для додатків, які вимагають надшвидкого доступу до даних, сховищ і маніпуляцій з ними, а також в системах, які не мають жорсткого диску, але, тим не менш, повинні управляти значною кількістю даних.

Згідно зі звітом McObject, in-memory СУБД відмінно масштабуються за розміри, що перевищують терабайт. Так, в ході проведених тестів 64-бітна in-memory СУБД встановлена на 160-ядерному сервері SGI Altix 4700 під керуванням SUSE Linux Enterprise Server версії 9 від Novell досягла 1,17 терабайт і 15,54 млрд рядків без видимих обмежень для подальшого масштабування. Причому продуктивність в даному тесті практично не змінювалася в міру того,

як СУБД досягла сотень гігабайт, а потім і терабайта, що свідчить про практично лінійну масштабованість.

In-memory СУБД може бути як «вбудованою СУБД», так і «клієнт-серверною». Клієнт-серверні СУБД по суті своїй розраховані на велику кількість користувачів, так що in-memory СУБД також можуть бути розділені на кілька потоків / процесів / користувачів.

IMDG має ряд переваг перед реляційними БД:

1. ***Швидкодія***. Всі дані знаходяться в оперативній пам'яті кластера, за рахунок чого істотно скорочується час доступу.

Оскільки всі дані серіалізовані, то час отримання будь-якої об'єкт з кешу = (час переміщення об'єкта на конкретному вузол кластер) + (час на десеріалізацію).

У випадку, якщо запитуваний об'єкт знаходиться на тому ж вузлі, на якому виконано запит, то (час отримання) = (час на десеріалізацію). І тут ми бачимо, що доступ до даних міг би бути взагалі безкоштовним, якби об'єкт не треба було б десеріалізовувати, для чого в концепцію ММОГ було введено поняття Near-cache.

Near-cache – це локальний кеш об'єктів для швидкого доступу, всі об'єкти в ньому зберігаються готові до використання. Якщо біля кешу для даного кешу конфігурується, то об'єкти туди попадають автоматично при першому отриманні запиту цих об'єктів.

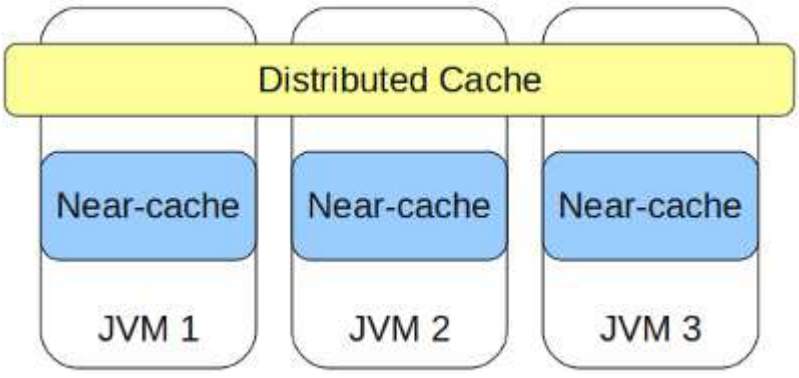

Рисунок 1.4. Локальний кеш об'єктів для швидкого доступу

2. ***Надійність***. Дані в кластері зберігаються в партіції (частинах), і ці партіції рівномірно розподілені по кластеру, а кожна розбиття реплицируется на деяку кількість вузлів (в залежності від конфігурації кластера і від вимог до надійності зберігання даних). Попадання об'єкта в ту чи іншу партіцію однозначно визначається деякою хеш-функцією.

Так як при роботі під високим навантаженням вихід окремого вузла кластера (або декількох вузлів відразу, якщо це були віртуальні машини всередині одного залізного сервера) не є чимось неймовірним, то для забезпечення схоронності даних в конфігурації кеша вказується кількість вузлів, втрату яких кластер повинен безболісно пережити. Цей показник визначає кількість копій кожної партіції.

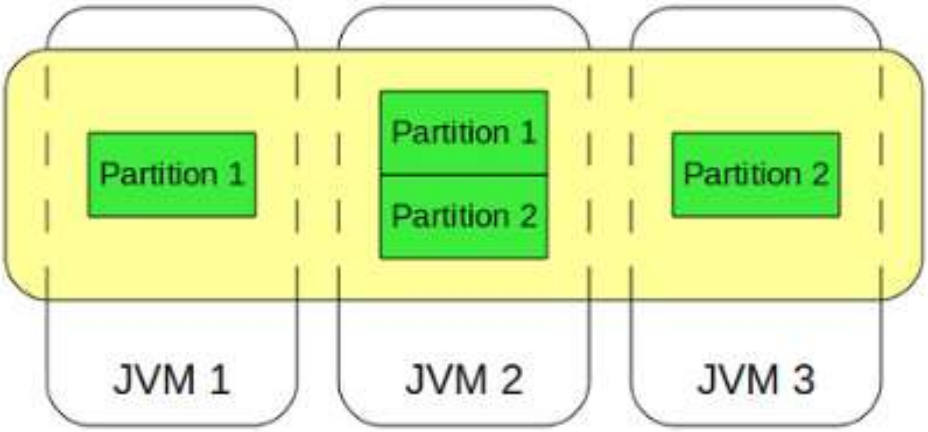

Рисунок 1.5. Партіції в кластерах

Тобто якщо ми вкажемо, що втрата 2 вузлів не повинна привести до втрати даних, то кожна розбиття буде зберігатися на 3 різних вузлах кластера, і при падінні 2 вузлів дані залишаться неушкодженими. Якщо при цьому в кластері залишилося більше одного вузла, то знову буде створено 3 копії всіх даних, і кластер буде готовий до нових неприємностей.

3. ***Масштабованість***. Склад кластера (кількість вузлів) може змінюватися без зупинки роботи всього кластера, а за коректною роботою кластера і

консистентним і доступністю даних стежить сам grid без будь-якого втручання програміста. Тобто при зростаючій навантаженні або обсязі даних, ви можете просто підняти ще кілька сконфігурованих вузлів, які автоматично приєднаються до кластеру, а дані всередині самого кластера перебалансірует для рівномірного розподілу даних по вузлах, при цьому обсяг переміщуються даних буде мінімальний, щоб не створювати зайве навантаження на мережу .

4. ***Актуальність даних***. При використанні IMDG ви завжди отримуєте актуальні дані, тому що при виконанні put надсилається повідомлення всім вузлам кластера про те, що об'єкти з такими-то ключами отримали нове значення. Кожен вузол оновлює свої партіції, що містять ці ключі, і видаляє старі значення зі свого near-cache.

5. ***Зниження навантаження на БД***. IMDG можна використовувати не тільки як самостійне сховище, а й як вузол системи, що знімає навантаження з реляційної БД із її складнощами у масштабуванні.

Розглянемо звіт компанії Aberdeen Research щодо впровадження технології IMDG. У цьому звіті подана інформація щодо реалізації вказаної технології в обробці великих даних, визначено існуючі труднощі в обробці і аналізі зростаючого обсягу даних, проаналізовано, яким чином обчислення в пам'яті можуть відігравати ключову роль у прискоренні збору, спільного використання інформації на підприємстві та управління нею.

У звіті зазначено, що щороку обсяг бізнес-даних зростає на 36%. Із 196 клієнтів Aberdeen, які здійснюють обробку Великих даних, 89 використовують обчислення в пам'яті. Причина, яка ще недавно зумвлювала відмову більшості компаній від цієї технології, полягає, швидше за все, в її високій вартості.

Отримання інформації за запитом займає 42/75 часу, порівняно із його витратами при використанні звичайних технологій.

При обчисленнях в пам'яті обробляється 1200 Тб / ч, в порівнянні з 3,2 Тб при використанні звичайних технологій. У наявності підвищення ефективності в 375 разів.

Обчислення в пам'яті, простіше кажучи, роблять швидкими обробку і аналіз інформації, і це добре для користувачів та IT-організацій, що мають справу з постійно зростаючими обсягами інформації, що використовується при прийнятті бізнес-рішень.

Проте, необхідно відзначити, що обчислення в оперативній пам'яті, як і будь-яка технологія, мають свої унікальні особливості, проблеми та підводні камені. По-перше, це досить велика вартість, яка тільки останнім часом зменшується завдяки різкому падінню цін на носії RAM. Потрібні потужні сервери, багатоядерні процесори і тонни оперативної пам'яті. Необхідно відповідне ПЗ і аналітичні програми. Технологія швидкісної обробки вимагає застосування всіх перерахованих компонентів, оскільки терабайти даних зберігаються з «нульовою» латентністю доступу безпосередньо в оперативній пам'яті серверів, а не десь на дисках.

Інша проблема технології обчислень в оперативній пам'яті полягає в тому, що вона добре підходить тільки для транзакцій з наборами структурованих даних, таких як артикули товарів, інформація про покупців, звіти з продажу.

Якщо компанія має в своєму розпорядженні засоби та розуміє цінність інформації в сучасній бізнес-стратегії, технологія обчислень в оперативній пам'яті може стати для неї підходящим вибором.

Серед рішень, які на ринку використовуються найчастіше – рішення Oracle, IBM Cognos TM1, SAP HANA, Microsoft PowerPivot, QlikView і Pentaho Business Analytics. Такі платформи добре використовувати при необхідності аналізу даних в режимі реального часу з урахуванням того, що в процесі аналізу дані можуть бути змінені в будь-який момент. Також такі рішення добре підходять у випадках, коли немає можливості створення багатовимірного

сховища даних і потрібно проводити аналіз даних облікової системи без її модифікації. Системи пропонують різні способи горизонтального масштабування, як з використанням засобів самої платформи, так і з застосуванням додаткового ПЗ.

Зокрема, функцію роботи з віртуальними кубами в Pentaho Business Analytics можна масштабувати з використанням промислового рішення JBoss Data Grid, яке призначене для створення розподілених in-memory сховищ інформації.

За такого підходу можна створювати in-memory куби об'ємом 1 Тб і вище. З точки зору цінової доступності, ці рішення цілком задовільні для SMB компаній. Зокрема, для SMB ринку у IBM є комплексне рішення Cognos Express (TM1 є її частиною), а у Pentaho є безкоштовна версія і спеціальні цінові пропозиції для невеликих компаній.

Строго кажучи, технології in-memory діляться на два класи. Це рішення data discovery і саме in-memory, або, як правильніше, «системи управління базами даних (СУБД) in-memory». Приклад рішення data discovery – Qlikview. Дані в цій системі представлені в зручній формі, а використання технології in-memory дозволяє швидко працювати із візуальною складовою. Але до неї не можна підключити інші інструменти: дані з файлів Microsoft Excel, систем Cognos або Oracle BI.

Приклад рішення in-memory СУБД є SAP HANA. Ця система, будучи СУБД, надає доступ до пам'яті будь-якого BI-інструменту: можна завантажити дані з таблиць Excel, систем Cognos, Oracle BI і інших.

Для ефективного використання грід потрібно не тільки розподілити дані по вузлах для зберігання, але ще і обробляти їх на тих же вузлах, де лежать дані. Цю задачу вирішують IMCG, які від IMDG відрізняються орієнтацією на виконання розподіляються за тими ж вузлів кодів. Тісна інтеграція між IMDG і IMCG грунтується на тому, що вони представляють дві сторони однієї медалі.

Функціональність IMCG можна розділити на наступні групи.

- Впровадження та контроль використання (Distributed Deployment & Provisioning). Незручність і великі трудовитрати при виконанні цих дій завжди були слабкою стороною грід, що ускладнювало їх використання, а іноді робило це неможливим. З часів перших систем Грід (Globus, Grid Engine, DataSynapse, Platform Computing) і до Hadoop в більшості проектів впровадження і внесення серйозних змін були пов'язані з великим обсягом ручної роботи з перебудови бібліотек і повторного запуску всіх сервісів. Розподіленість архітектури IMCG підсилює цю проблему, тому відповідні технології, наприклад GridGain, включають засоби, що знижують трудовитрати.
- Розподілене управління ресурсами (Distributed Resources Management). Сюди входить управління фізичними пристроями, віртуальними машинами і операційними системами. У IMCG однією з найважливіших функцій є автоматична підтримка топології і виявлення змін. Узгодженість топології необхідна для того, щоб всі вузли «бачили» всі зміни (вихід з ладу вузлів, підключення нових) одноманітно.
- Розподілені моделі виконання (Distributed Execution Models). Ці моделі і перетворюють IMCG в обчислювальну інфраструктуру. Таких моделей може бути кілька, і найбільш поширена відноситься до класу «розгалуження з об'єднанням» (fork-join type) або MapReduce - поділ завдання на безліч підзадач по будь-якою ознакою, а потім їх об'єднання у вигляді підсумкового результату. Найбільш популярна реалізація MapReduce, відома як Hadoop, повністю орієнтована на роботу з дисками, а MapReduce в реалізації GridGain робить приблизно те ж саме, але в оперативній пам'яті. У моделях виконання підтримуються архітектури з масовим паралелізмом (Massively Parallel Processing, MPP) і віддалений виклик процедур (Remote Procedure Call, RPC), а

також розподілені обчислення з обміном повідомленнями, потокові обчислення і обробка складних подій.

Вартість таких рішень складається з великої кількості факторів, починаючи від термінів впровадження проекту і закінчуючи вартістю самої технології. Деякі рішення дійсно коштують дорого, але швидко окупаються за рахунок підвищення ефективності та оперативності роботи. Такі продукти затребувані в будь-якій компанії, де важливо отримувати аналітичні звіти оперативно. Наприклад, в разі, якщо аналітику потрібно сформувати звіт, що складається з 30 Excel файлів, йому буде потрібно мінімум 3 дні для того, щоб звести його вручну. При наявності необхідних IT-систем, йому достатньо лише вказати на ці 30 файлів, після чого система сама сформує єдиний звіт, з яким можна буде працювати.

1.3 Методи реалізації розподілених сховищ даних та механізмів завантаження, що в них використовуються

Технологія IMDG вже активно використовується, і на її основі виникли такі сервіси, як VmWare Gemfire (Apache Geode), Oracle Coherence, Hazelcast, MemSql, Couchbase та інші.

Hazelcast - це опенсоурс IMDG. Він забезпечує масштабоване розподілене обчислення в пам'яті, яке широко визнається як найшвидший та найбільш масштабований підхід до продуктивності додатків. Що важливіше, Hazelcast спрощує розподілені обчислення, пропонуючи розподілені реалізації багатьох дружніх для розробників інтерфейсів з Java, таких як Map, Queue, ExecutorService, Lock та JCache. Наприклад, інтерфейс Map надає сховище даних в пам'яті, яке надає багато переваг NoSQL з точки зору зручності для розробників та продуктивності розробників. На додаток до розподілу даних In-Memory,

Hazelcast надає зручний набір API для доступу до процесорів кластера для максимальної швидкості обробки. Hazelcast розроблений легким і простим у використанні. Оскільки Hazelcast поставляється як компактна бібліотека (JAR), і оскільки у нього немає зовнішніх залежностей, крім Java, він легко підключається до програмного рішення і надає розподілені структури даних і розподілені обчислювальні утиліти. Hazelcast відрізняється високою масштабованістю і доступністю. Розподілені додатки можуть використовувати Hazelcast для розподіленого кешування, синхронізації, кластеризації, обробки, обміну повідомленнями pub / sub і т. д. Hazelcast реалізований на Java і має клієнти для Java, C / C ++, .NET і REST. Hazelcast також розуміє протокол memcache. Він підключається до Hibernate і може бути легко використаний з будь-якою існуючою системою баз даних [3].

MemSQL - це розподілена реляційна база даних, яка обробляє змішані транзакції і аналітику в режимі реального часу в масштабі. Він доступний через стандартні драйвери SQL і синтаксис і підтримує широку екосистему драйверів і додатків. MemSQL має дворівневу архітектуру, яка забезпечує високу пропускну здатність. Це розподілена система, яка може масштабуватися по горизонталі на товарному обладнанні і дуже сумісна з іншими технологіями в сучасній екосистемі обробки даних (наприклад, на платформах оркестровки, середовищах розробників і інструментах BI). Він оснащений вбудованим сховищем рядків і дисковим сховищем на диску. Він також оснащений Streamliner, інструментом, який може ефективно передавати дані в сховище рядків і стовпців MemSQL [4].

Galaxy - це високопродуктивна матриця даних у пам'яті (IMDG), яка може служити основою для побудови розподілених додатків, що вимагають точного управління розміщенням даних і / або розподілених структур даних, що настроюються. Що відрізняє Galaxy від інших IMDG, це спосіб привласнення елементів даних вузлу кластера. Замість розбиття даних по одному з ключів з

використанням узгодженої схеми хешування, Galaxy динамічно переміщує об'єкти з одного вузла на інший в міру необхідності, використовуючи протокол кеш-когерентності, аналогічний протоколу, знайденому в процесорах. Це робить Galaxy придатним для додатків з передбачуваними шаблонами доступу до даних, тобто додатками, де елементи даних поводяться відповідно до деякої метрики близькості, а елементи які «ближче», швидше за все, будуть доступні разом, ніж елементи, які «далеко» один від одного [5].

Як і будь яка технологія, IMDG має свої недоліки, подолання яких на даний час є актуальним завданням. До таких недоліків відносяться:

енергозалежність;

переповнення оперативної пам’яті;

блокування.

Енергозалежність. Найбільшим недоліком IMDG з фізичної точки зору звичайно є їх енергозалежність. Основним накопичувачем інформації є оперативна пам’ять, яка не є довготривалим запам’ятовуючим пристроєм (ЗП). Відповідно навіть невеликий збій електромережі може призвести до втрати даних.

Найпростішим способом вирішення проблеми енергозалежності є механізм резервного копіювання даних на більш надійні довгочасних ЗП. Наприклад, при синхронному копіюванні операція додавання/зміни/видалення даних вважається завершеною лише після підтвердження інформації від довгочасного ЗП про вдале виконання цієї операції на ньому.

Також резервну копію можна обновлювати в асинхронному режимі. При цьому запис на довгочасний ЗП виконується у фоновому режимі, а підтвердження про вдале завершення операції повертається користувачу одразу після модифікації основного сховища в ОЗП [1].

Резервне копіювання дозволяє уникнути втрати даних при відмові будь-якої кількості вузлів сховища, але для відновлення з резервної копії після відмови необхідно перезапустити все сховище.

Іншим способом вирішення проблеми енергозалежності є створення копій даних на різних вузлах розподіленої системи. При цьому будь-яка модифікація виконується на декількох вузлах одночасно. Таким чином, різні вузли мають мати різне електропостачання щоб при відмові одного з вузлів, вся інформація збереглася на іншому. Відповідно перезапуск системи в даному випадку не потрібний, але все ж з'являються відчутні втрати швидкості передачі всередині мережі.

Переповнення оперативної пам'яті. Як відомо об'єм оперативної пам'яті є значно меншим ніж об'єм пам'яті на довгочасних ЗП. Відповідно зі збільшенням сховища, не виключено повне заповнення вільного місця в ОЗП.

Одним із варіантів вирішення цієї проблеми є перенесення даних з ОЗП на довгочасний ЗП, але при цьому продуктивність роботи сховища зводиться до мінімуму, відповідно цей варіант не є допустимим. Це означає, що переповнення в розподіленому сховище необхідно уникати.

Для уникання таких ситуацій існує декілька методів:

Data Expiration (старіння даних) – кожному об'єкту в кеші заздалегідь назначається певний час життя, після якого він буде видалений. Таким чином в оперативній пам'яті будуь існувати лише актуальні дані;

Data Eviction (вигнання даних) – це механізм, що запобігає допущення переповнення в оперативній пам'яті. Його суть у тому, що для кожного вузла сховища визначається певна критична грань використання пам'яті, після якої частина даних видаляється з основної пам'яті. Щоб уникнути втрати нових і актуальних даних, видаляються дані за певним заздалегідь заданим алгоритмом: або найбільш старі дані, або дані, що найменше використовуються. Також існує можливість видалення найбільш об'ємних даних [3].

Блокування. Як вже було сказано, для збереження узгодженості модифікованих даних в сучасних розподілених сховищах використовується механізм блокування. Він блокує модифікацію даних, до моменту отримання підтвердження завершення попередньої операції на всіх вузлах сховища. Про те, цей механізм значно знижує продуктивність сховища при великій кількості операцій запису, формуючи чергу для запитів що стосуються модифікації даних. Ця проблема на даний момент є актуальною.

Для читання і запису з / в БД для кожного кеша в конфігурації вказується Loader, який буде відповідати за читання / запис об'єктів в БД.

Можливі кілька варіантів організації доступу до даних:

під час запуску програми витягувати всі необхідні дані з БД в грід (так званий preloading). Час підйому додатки збільшується, споживання пам'яті теж, але швидкість роботи зростає

під час роботи програми підтягувати необхідні дані по запитах клієнтів (read-through). Виконується автоматично за допомогою об'єкта Loader для даного кешу. Час завантаження додатків досить малий, початкові витрати пам'яті теж, але виникають додаткові тимчасові витрати на обробку запитів, що викликають read-through.

Варіанти запису в БД при зміні відповідних даних:

при кожній операції put в кеш автоматично робиться запис в БД за допомогою Loader'a (так званий write-behind). Підходить тільки для систем, основне навантаження на які викликається читанням.

дані, які очікують запису в БД, накопичуються, а потім проводиться один запит на запис в БД. Сигналом до виконання такого запиту може бути певна кількість даних, які очікують запису, або таймаут. Підходить для write-intensive систем, але складніше в реалізації

У разі використання IMDG як вузла, який бере на себе все навантаження з читання / запису / розподіленої обробки даних, ми продовжуємо мати в БД

актуальні дані, низьке навантаження на саму базу і, що є дуже важливим моментом, корпоративні програми, що використовують БД для збору статистики, складання звітів і т.д. продовжують працювати в колишньому стані.

В такому ракурсі, важливого значення набувають класи ресурсів. Класи ресурсів являють собою визначені обмеження ресурсів, що регулюють виконання запиту. Сховище даних SQL обмежує обчислювальні ресурси для кожного запиту відповідно до класу ресурсів.

Класи ресурсів допомагають керувати загальною продуктивністю робочого навантаження сховища даних. Використання класів ресурсів значно спрощує управління робочим навантаженням за допомогою завдання обмежень на кількість паралельно виконуються запитів, а також обчислювальних ресурсів, призначених для кожного запиту.

Невеликі класи ресурсів використовують менше обчислювальних ресурсів, але забезпечують більший паралелізм запитів.

Великі класи ресурсів використовують більше обчислювальних ресурсів, але забезпечують менший паралелізм запитів.

Класи ресурсів призначені для управління даними та інших дій з управління. Їх також можна використовувати для деяких дуже складних запитів при наявності множинних об'єднань і сортування, коли система виконує запит в пам'яті, а не на диску.

Класами ресурсів регулюються такі операції:

INSERT-SELECT, UPDATE, DELETE;

SELECT (при запиті таблиць користувача)

ALTER INDEX - REBUILD або REORGANIZE;

ALTER TABLE REBUILD

CREATE INDEX

CREATE CLUSTERED COLUMNSTORE INDEX

CREATE TABLE AS SELECT (CTAS)

завантаження даних

Операції переміщення даних, здійснювані службою Data Movement Service (DMS)

Статичні і динамічні класи ресурсів. Існує два типи класів ресурсів: динамічний і статичний.

Статичні класи ресурсів виділяють один обсяг пам'яті незалежно від поточного рівня обслуговування, вимірюваного в одиницях використання сховища даних. Це статична виділення означає, що на розширених рівнях обслуговування ви можете виконати більше запитів в кожному класі ресурсів. Статичні класи ресурсів називаються staticrc10, staticrc20, staticrc30, staticrc40, staticrc50, staticrc60, staticrc70 і staticrc80. Вони більше підходять для рішень, що підвищують клас ресурсу для отримання додаткових обчислювальних ресурсів.

Динамічні класи ресурсів виділяють змінний обсяг пам'яті в залежності від поточного рівня обслуговування. При збільшенні масштабу рівня обслуговування запити автоматично отримують більший обсяг пам'яті. Динамічні класи ресурсів називаються smallrc, mediumrc, largerc і xlargerc. Вони більше підходять для рішень, що збільшують масштаб обчислювальних ресурсів для отримання додаткових ресурсів.

Важливим аспектом In-Memory Data Grid (IMDG) реалізації розподілених сховищ даних є динамічне завантаження класів.

Динамічне завантаження – це завантаження під час виконання певного процесу.

Будь який клас, що використовується в середовищі виконання, був так чи інакше туди завантажений будь-яким завантажником Java. Зазвичай класи завантажуються за необхідністю їх використання. Окрім цього базові класи завантажуються під час старту програми.

В мові програмування Java існує 3 види завантажників:

1) Базовий завантажник (bootstrap);

2) Системний завантажник (system classloader);

3) Завантажник розширень (extension classloader) [7].

Bootstrap – реалізований на рівні віртуальної машини Java (JVM) і не має зворотнього зв’язку із середовищем виконання. Цим завантажником завантажуються саме базові класи Java. Відповідно доступу до нього із керованого середовища немає. Керувати завантаженням базових класів можливо лише за допомогою ключа особливого ключа (-Xbootclasspath), який дозволяє перевизначити набори базових класів.

System Classloader – системний завантажник, реалізований на рівні середовища виконання (Java Runtime Environment, JRE). Цим завантажником завантажуються класи, що вказані в змінних середовища CLASSPATH. Керування цими класами можливо за допомогою ключа –classpath, або за допомогою системних опцій.

Extension Classloader – завантажник розширень. Керування цим завантажником можливо лише за допомого системних опцій.

Щоб створити свій власний завантажник класів, необхідно щоб він розширював стандартний Classloader.

Існує два варіанти завантаження класів:

Статичне завантаження класів – це звичайне завантаження, що проводиться автоматично. При старті програми завантажник класів рекурсивно завантажує усі класи, що зустрічаються у програмі починаючи з основного класу. Об’єкти таких класів створюються стандартним способом за допомогою оператора new;

Динамічне завантаження класів відтворюється за допомогою використання Classloader. Динамічне завантаження має сенс, коли необхідно завантажити клас під час виконання програми, коли необхідно замінити клас, змінив якусь певну логіку, і коли при цьому перезапуск програми не є раціональним.

У наш час проблема динамічного завантаження все більше набуває популярності. Якщо розглядати розподілені сховища даних в оперативній пам’яті, слід розуміти, що при змінах у певному класі, необхідно повністю перезапускати систему, що призводить до неабияких труднощів [7].

У загальному вигляді система динамічного завантаження класів у розподілених сховищах даних представлена на рисунку 1.7.

Рис. 1.6. Система динамічного завантаження класів у розподілених сховищах даних

Відповідно почали з’являтися рішення, що так чи інакше вирішують проблему динамічного завантаження та оновлення.

Hot Swap і Hot Deploy – це бібліотека, представлена стандартним відладчиком Java. Очевидним недоліком є те, що неможливо змінювати структуру класу, а саме додавати, змінювати методи або поля, відповідно ця технологія не є досить гнучкою для використання.

Hot Deploy – технологія що представлена контейнерами сервлетів. Суть заключається у тому, що необхідно просто завантажити war-файл на сервер, і програма запускається заново. У цього способу також є очевидний недолік: по-перше це призупинення серверу повністю, а відповідно певний час він буде недоступним, і це не задовольняє вимоги. А по-друге, якщо десь було допущено помилку, то у цієї технології немає можливості контролю за якимись певними змінами, і відповідно програму необхідно буде запускати заново [7].

JRebel – це інструмент, що створений саме для внесення змін без перезапуску програми. При додаванні методу в коді, JRebel викликає extract method, що призводить до додавання нового методу до класу. Як вже було зазначено, стандартна технологія Hot Swap не зможе додати такі зміни [7].

JRebel дає можливість завантажувати ресурси напряму із робочого середовища, відповідно необхідно лише скомпілювати змінений код, що займає набагато менше часу, ніж при повному зборі архіву.

JRebel не перезапускає програму та не створює нові завантажники класів, і відповідно ризик отримати витоки пам’яті зменшується. Стан об’єктів на сесії зберігається незмінним, відповідно щоб побачити зміни, необхідно лише оновити сторінку.

Ще одним плюсом є те, що ця технологія підтримується напряму з сучасних середовищ розробки, і відповідно більш зручна.

Про те, як і у всіх технологій є певні технічні обмеження.

Наприклад при зміні ієрархії класів, змінити його неможливо. Також JRebel намагається зберігати стан об’єктів, що уже створені у пам’яті, відповідно перезапуск конструкторів не відбувається. Це призводить до наступних проблем: змінивши значення статичного поля, у новій версії класу воно повинно бути оновлене, але значення це буде присвоєне у статичному блоці, який JRebel не перезапустив, відповідно нове значення не буде ініцалізовано. JRebel оновить статичний блок лише у випадку додавання нового статичного поля. Причина цього явища полягає в тому, що те, що відбувається в статичному блоці може вплинути на стан об’єкта, і тому перезапуск відбувається лише в крайньому випадку.

Ще однією проблемою через не перезапуск конструкторів є те, що при додаванні нового поля в класі буде присвоєно значення “за замовчуванням” для даного типу. Відповідно при додавання поля з типом, що не є примітивом, то присвоєне значення автоматично стане null.

Рішення від ScaleOut. На рис. 1.7. представлена ідеальна схема того, як за версією ScaleOut повинна працювати аналітична система на базі IMDG.

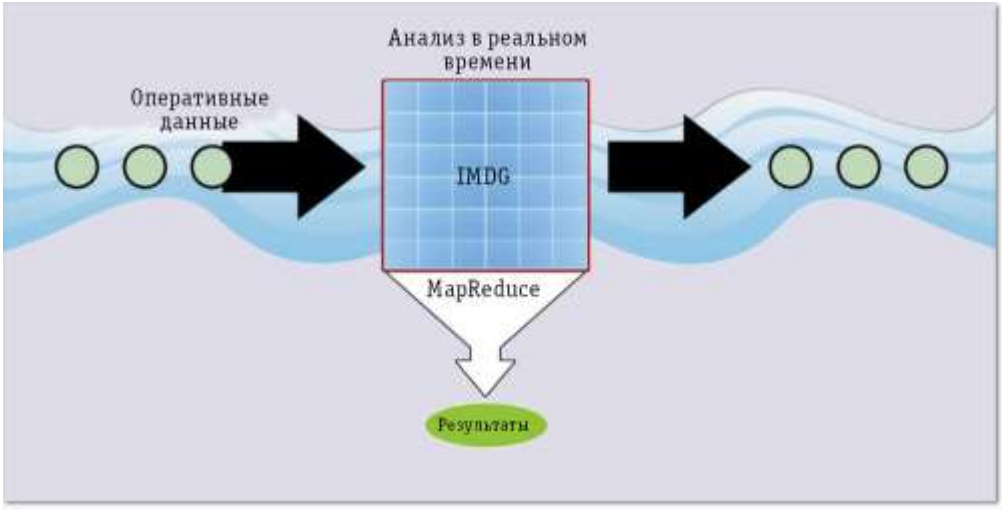

Рис. 1.7. Дані на потоці в IMDG.

Грід стоїть «на потоці» (in-bound) швидкоплинних оперативних даних і виділяє ті чи інші суттєві ситуації відразу при їх виникненні. Це може бути,

наприклад, перевірка платіжних операцій безпосередньо по ходу їх виконання або будь-яке інший додаток, де потрібна аналітика, синхронна з потоком даних. Якщо така система працює, то можна майже миттєво виявляти критично важливі події і вживати превентивних заходів. Традиційна аналітика Великих Даних на це поки не здатна і розраховує на збереження даних в розподіленої файлової системи і потім на їх обробку в пакетному режимі. Навіть перехід на твердотільні накопичувачі дозволить зменшити час виконання пакету завдань, але реальний час все одно залишиться недосяжним. Рішенням може бути аналітична машина на базі IMDG, що перетворює просте розподілене по вузлах кластера сховище даних в паралельну платформу аналізу даних в реальному часі. Завдання власне платформи IMDG полягає в автоматичному балансуванні навантаження, в мінімізації переміщення даних і організації роботи з даними за місцем їх знаходження.

Аналітичний сервер ScaleOut Analytics Server являє собою типовий приклад IMDG, поєднаний з машиною для аналізу даних (рис. 1.8.). Дані в машину надходять з двох типів джерел, що розрізняються за вимогами до затримки: якщо затримка повинна бути мінімальною, то працює додаток динамічно безперервно оновлює грід в темпі надходження даних; якщо вимоги не такі жорсткі, то дані можуть затриматися в системі зберігання. У будь-якому випадку IMDG утримує дані, для надійності створює їх репліки і розподіляє по серверам. ScaleOut Analytics Server дозволяє створювати специфікацію запитів або з використанням фільтрів на Java, або на мові інтегрованих запитів Microsoft LINQ, якщо використовується C#. Ці специфікації дозволяють вибирати аналізовані дані за тими чи іншими ознаками.

Рисунок 1.8. Аналітична машина MapReduce IMDG Scaleout

У ScaleOut Analytics Server прийняті в Hadoop методи аналізу і злиття (analysis і merge) можуть бути написані або на Java, або на C#. Оскільки IMDG зберігає об'єкти, що аналізуються та зливаються, в оперативній пам'яті, то ці методи можуть бути написані без використання прикладного інтерфейсу API до грід. В такому випадку аналітичний метод специфікує логіку аналізу в додатку до одного об'єкту, обирається специфікацією запиту. Наприклад, у разі біржі це може бути обробка показників однієї компанії, а злиття дає узагальнені показники по групі об'єктів. На додаток до ScaleOut Analytics Server існує спеціальний API, так званий «виклик» (invoke), і додаткова специфікація запитів для методів аналізу і злиття, що дозволяє запускати модель обчислення, що називається «паралельний метод виклику» (Parallel Method Invocation, PMI). В такому випадку IMDG автоматично виконує запити, аналіз і злиття паралельно на всіх своїх серверах, утворюючи машину мультипотокового обчислення. Таким чином ScaleOut Analytics Server дозволяє уникнути переходу в пакетний режим і наблизитися до реального часу.

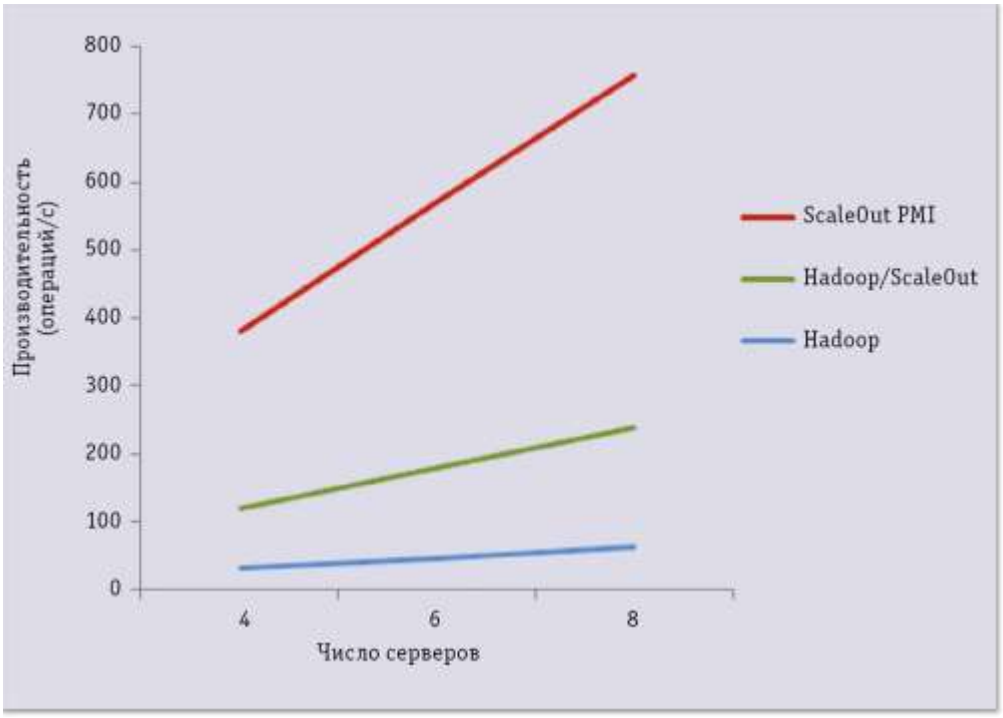

Рисунок 1.9. Порівняння продуктивності

Описана вище схема спрощує трудомісткий процес програмування Hadoop, при якому навіть досвідченому програмісту потрібно кілька тижнів на освоєння специфічних прийомів. Використання функцій ScaleOut Analytics Server відкриває можливість для розробки аналітичних додатків фахівцям, які є професіоналами в прикладних областях.

На рис. 1.9. наведено порівняння продуктивності різних реалізацій аналітичної системи на базі IMDG.

Рішення від GridGain. У найбільш загальній формі підхід до IMDG, якого дотримуються в компанії GridGain, ілюструє рис. 1.10. Конфігурація складається з множини серверів, що працюють In-memory і передають дані та коди інших серверів. При масштабуванні число серверів, що входять в грід, може перевищувати тисячу. За необхідності грід може отримувати дані і коди з традиційних баз даних, NAS і SAN. В цілому платформа від GridGain поєднує в собі ПЗ сполучного шару, високошвидкісну реалізацію MapReduce і включає підтримку обчислювальні грід (GridGain Compute), грід даних (Data Grid) і

Великих Даних (за рахунок інтеграції файлової системи Hadoop Distributed File System і СУБД HBase). На базі платформи створюється три групи продуктів: In-memory HPC, In-memory Streaming і In-memory Database, а також прискорювачі Accelerator for Hadoop і Accelerator for Mango DB. При цьому In-memory HPC підтримує чотири моделі виконання: MapReduce Processing, RPC Processing, MPP Processing і MPI-Style Processing. Рішення з групи In-memory Streaming призначаються для обробки потоків, інакше званої обробкою складних подій.

GridGain IMDB це засноване на Java об'єктне сховище даних з доступом за ключами, яке можна представити у вигляді сукупності одного або декількох кешів, що називаються картами або словниками. Кожен кеш є не що інше, як розподілений набір пар ключ-значення уявлення об'єктів будь-якого певного типу користувачем.

Кожен кеш повинен бути заздалегідь налаштований – «на льоту» зробити це неможливо через необхідність підтримувати гармонію між розподілених семантик. При роботі з GridGain IMDB у вбудованому режимі кеші і їх проекції служать основними вхідними точками API. Хоча кожен кеш має безліч конфігураційних опцій, всі кеші поділяються тільки на три типи або режими: локальний, реплікаційний і такий, що ділиться на розділи. У локальному режимі всі дані зберігаються без будь-якого розподілу, що вимагає організації обміну в пам'яті. При реплікації по всіх вузлах кластера забезпечується високий рівень готовності та надійності, що призводить до зайвих витрат ресурсів пам'яті. Третій режим відрізняється тим, що в кожному з вузлів зберігається невелика порція загальних даних, тому забезпечується найбільшу швидкодію.

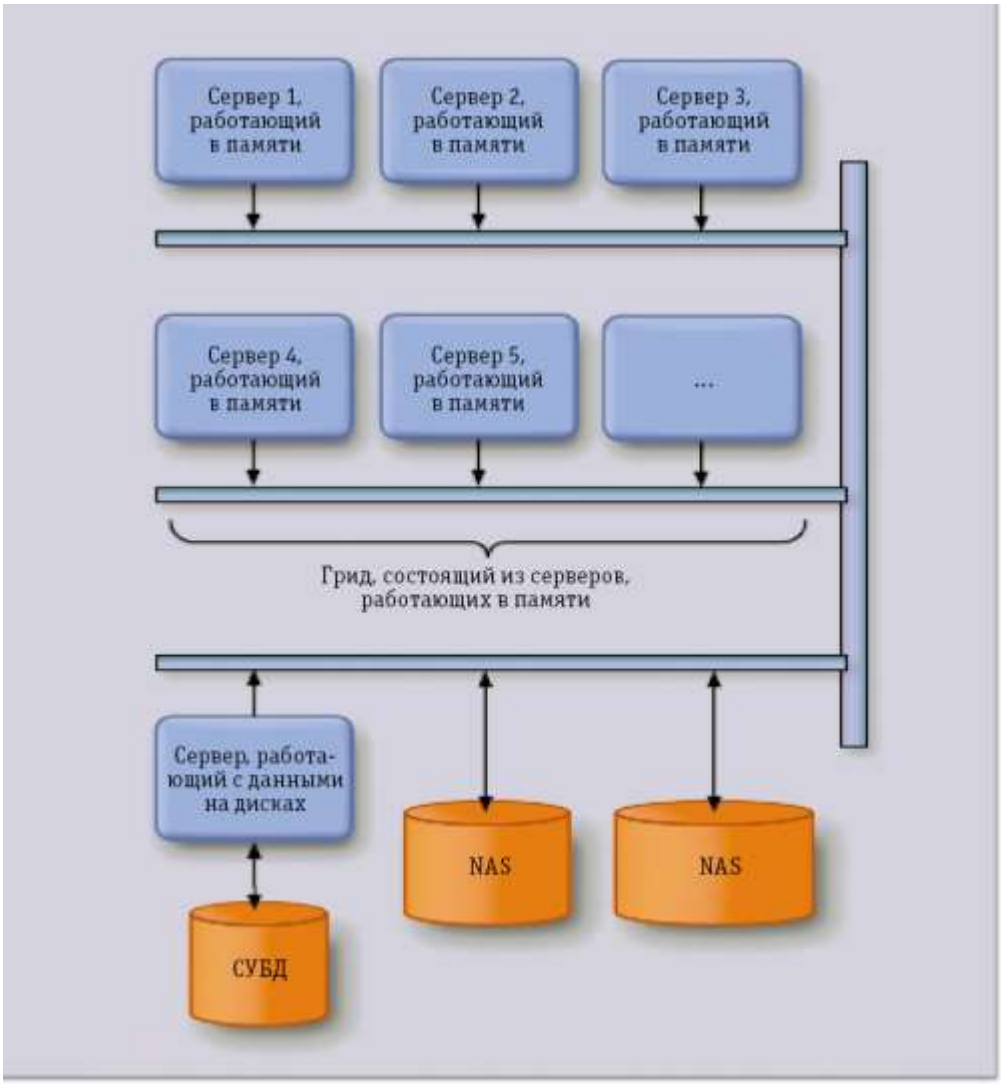

Рисунок 1.9. Архітектура IMDG від GridGain

Можливість комбінації всіх трьох режимів для кожного вузла не обмежує користувача якоїсь однієї моделлю зберігання і робить GridGain IMDB зручним розподіленим сховищем даних. GridGain IMDB зберігає дані на чотирьох системних рівнях (за зростанням часу затримки): в пам'яті – JVM on-heap, в пам'яті поза купи - JVM off-heap, в просторі підкачки локального диска або в додатковому кеші. Терміном heap, буквально «купа», позначають область пам'яті для динамічного розміщення об'єктів (кешу гріда і інших компонентів) - в разі Java heap об'єкти зберігаються в «купі», а в разі off-heap - поза нею. При розміщенні даних за межами купи JVM про це не знає - отже, не сповільнюється.

Істотною відмінною рисою GridGain IMDB є те, що спочатку розроблялось високоякісне розподілене масштабоване середовище, якому пізніше додали рис повноцінної СУБД. Зазвичай буває інакше - готові бази даних адаптують до можливості розподілу. GridGain IMDB грунтується на технології HyperClustering, що дозволяє масштабувати систему аж до 1000 вузлів в одній транзакції топології. Кластеризація GridGain IMDB побудована по тимчасовій топології, а управління транзакціями підтримується механізмом MVCC (MultiVersion Concurrency Control - «управління конкурентним доступом за допомогою багатоверсійності»).

Також існують інші технології, які можуть бути використані лише при використанні певних допоміжних технологій розробки. Наприклад – Spring Loaded, що може існувати лише разом зі Spring. Нажаль, ця технологія також не дозволяє змінювати ієрархію класів та конструктори.

ВИСНОВКИ ДО РОЗДІЛУ 1

Підводячи підсумки щодо викладеного в розділі 1 матеріалу, можна впевнено зазначити, що розподілені сховища даних в останні впевнено посіли досить вагоме місце в сучасному світі інформаційних технологій. Концепція надшвидкої обробки даних відіграє важливу роль в задоволенні сучасних потреб. Через певну обмеженість ресурсів ця технологія є більш дорогою, проте більш ефективною для вирішення багатьох задач, які мають у своїй основі саме вирішення питань швидкодії та продуктивності.

У порівнянні з реляційними СУБД, що розраховані на величезну кількість інформації, концепція IMDG має більш вузьконаправлений профіль використання, у IMDG є помітні переваги: за продуктивністю – пам'ять набагато швидша за жорсткі диски, а прямий доступ позбавляє від спроб передбачення

при читанні; по гнучкості - структура даних з доступом по ключам представляє велику свободу розробникам програм, які можуть змінювати зв'язок між моделлю даних і додатком. Немає необхідності думати про контент або форматах - те, що потрібно, поміщається під ключем, воно ж і виходить назад.

Зберігання даних в IMDG – це лише половина функціоналу, необхідного для in-memory архітектури. Дані, що зберігаються в IMDG, необхідно обробляти паралельно та з високою швидкістю. Зазвичай in-memory архітектура секціонує дані в кластері за допомогою IMDG, а згодом виконуваний код відправляється саме на ті сервера, де знаходяться необхідні йому дані.

З програмної точки зору IMDG можна розглядати як розподілене сховище об'єктів з інтерфейсом, схожим на інтерфейс з узгодженою структурою даних (Concurrent HashMap) в Java. У сховищі містяться об'єкти з ключами, причому об'єкти можуть бути будь-якими, а не тільки масивами байтів або рядками, що дає гнучкість - можна зберігати точно ті об'єкти, які вимагає бізнес-логіка. Здатність роботи з об'єктами відрізняє IMDG від IMDB, від СУБД NoSQL і NewSQL, а IMDG взагалі відрізняє можливість розподілу по безлічі серверів. По суті, IMDG можна розглядати як карти кешу, при якому кожен ключ хешується на певному вузлі, і чим більше вузлів, тим більше розмір кешу. Завдання полягає в тому, щоб розмістити дані по вузлах таким чином, щоб локалізувати дані - виключити в процесі обробки обмін даними між вузлами. В ідеалі IMDG взагалі виключає міграцію даних і володіє стабільною топологією. За надійністю зберігання IMDG не поступаються реляційним СУБД: підтримуються вимоги ACID (Atomicy - «атомарность», Consistency - «узгодженість», Isolation - «ізольованість», Durability - довговічність) і, на відміну від NoSQL, дані завжди є несуперечливими.

Вирішення окремих проблем, на кшталт енергозалежності, не стандартизованості мови запитів, блокування та переповнення оперативної пам’яті – має провідну роль в поліпшенні in-memory data grid систем. Одна з

проблем розподілених сховищ даних в оперативній пам’яті є неможливість зміни схеми даних без перезавантаження кластеру, що призведе до втрати даних. Для вирішення цієї проблемі необхідно реалізувати динамічне завантаження класів. Жоден з існуючих інструментів не вирішує цю задачу повністю. Відповідно проблема динамічного завантаження є актуальною і вирішення цієї проблеми має суттєво вплинути на подальший розвиток IMDG технології.

РОЗДІЛ 2. Модель системи динамічного завантаження класів у розподілених сховищах даних

2.1 Модель розподіленого сховища даних з IMDG архітектурою

Сьогодні існує велика кількість різних як комерційних, так і не комерційних розроблених сховищ даних. Кожне з яких відповідає вимогам In-memory-data-grid (IMDG) архітектури, про те всі вони мають певні відмінності в тому, чи іншому функціональному просторі. Як вже було зазначено, існують рішення, що підтримують стандартні типізовані мови запитів накшталт SQL, по різному зроблені різні модулі, що відповідають за покращення основних характеристик сховища, таких як збільшення паралелізму, мінімізація мережевого обміну, зниження числа блокувань необхідних для здійснення різних транзакцій, тощо. Нажаль жодна з існуючих систем не реалізує динамічне завантаження класів, саме тому, було вирішено розробити інструмент, що вирішить це питання.

Типова для систем з архітектурою In-memory-data-grid схема роботи представлена на рис. 2.1. З деякими допущеннями за такою схемою працюють створені на різних мовах програмування існуючі рішення IMDG. Якщо серед них розглядати найбільш нині популярні, то необхідно виділити такі рішення, як:

Oracle Coherence – Java/C/.Net;

VMWare Gemfire – Java;

GigaSpaces – Java/C/.Net;

JBoss RedHat – Java;

Terracota – Java.

Всі ці рішення є комерційними, а отже програміст не має доступу для змінення основних функцій їх роботи, відповідно для зміни методології завантаження класів необхідно сховище даних, що має відкритий код та яке можливо використовувати на некомерційній основі. Таке рішення було

представлено компанією Apache зі своїм спрощеним аналогом до Gemfire – Apache Geode.

Рисунок 2.1. Схема роботи системи.

Основною особливістю Geode є те, що для некомерційних цілей його використання є абсолютно безкоштовним та має повністю відкритий код. Тому для реалізації динамічного завантаження було обрано саме цю систему.

Apache Geode – це не комерційне розподілене сховище даних, що дає змогу управляти даними в режимі реального часу, сумісний з різними додатками які мають велику інтенсивність обробки даних, що поширені нині в хмарних

архітектурах. Geode розділяє пам’ять, CPU, мережеві ресурси та опціонально локальний простір на кілька процесів, щоб керувати об’єктами та поведінкою програми.

Geode використовує динамічні методи реплікації та розділення даних для забезпечення високої доступності, підвищення продуктивності, масштабованості і відмовостійкості. На додаток до розподіленого контейнеру з даними, Geode представляє собою систему управління даними в оперативній пам’яті, що забезпечує надійність асинхронних подій та гарантовано доставляє повідомлення.

Розподілене сховище Geode складається з певної кількості членів-кешів. Кеши - це абстракції, що описують вузол у цьому розподіленому сховищі даних. Додатки що розробляються за допомогою Geode можуть мати різну топологію вузлів, що залежить безпосередньо від потреб архітектури програми: вузли можуть бути влаштовані за топологією рівноправних вузлів (peer-to-peer) або за топологією клієнт/сервер.

Всередині кожного кешу, є можливість виділити окремі області даних, так звані регіони. Області даних – це аналоги до таблиць, що використовуються в реляційних базах даних та управляють даними в розподіленому режимі у вигляді ім’я/значення. Реплікований регіон зберігає ідентичні копії даних на кожному члені кеші розподіленої системи. Роздільний регіон розподіляє дані між усіма членами. Після усіх налаштувань системи, клієнтські програми можуть отримати доступ до розподілених усередині регіонів без знань базової архітектури всієї системи. Також, за бажанням, є можливість визначити слухачів, які будуть повідомляти про зміну певних даних, і є можливість визначати критерії придатності даних, заради видалення застарілих даних з регіонів, для звільнення місця.

Для великих виробничих систем, Geode має механізм, що називається локатор. Локатори забезпечують як виявлення, так і балансування навантаження

різних сервісів всередині системи. В налаштуваннях клієнта необхідно задати список локаторів, а локатори, в свою чергу, зберігають динамічний список членів серверів. Для виявлення одне одного, серверам необхідно задавати єдиний порт, за замовчуванням це порт 40404.

Розглянемо архітектуру розподіленого сховища даних Geode. Як вже було зазначено, існує декілька варіантів топологій всередині Geode. Система може бути масштабована або по-горизонталі, або по-вертикалі.

Geode дає змогу використовувати різноманітні топології кешів:

В основі усіх систем лежить розподілена система рівноправних вузлів (peer-to-peer);

Для масштабування по-горизонталі або по-вертикалі, надається можливість об'єднання окремих систем в топологію клієнт/сервер або в WAN топологію.

У системах формату клієнт/сервер, невелика кількість серверних процесів управляє даними і обробляє події для набагато більшої групи клієнтів [9].

У WAN системах особливістю є те, що кілька географічно розрізнених систем слабо пов'язані в єдину, згуртовану в блок обробки.

Основною особливістю систем з топологією peer-to-peer, є те, що всі з'єднані вузли в середині системи – рівноправні. Комп'ютерні мережі типу peer-to-peer засновані на принципі рівноправності учасників і характеризуються тим, що їх елементи можуть зв'язуватись між собою, на відміну від традиційної архітектури, коли лише окрема категорія учасників, що називається серверами може надавати певні сервіси іншим. Іншими словами, в цій топології кожен вузол підтримує канали зв'язку для кожного іншого вузла, що дозволяє одній мережі доступатись до будь якого іншого члена кешу. Таким чином, навіть якщо клієнт (що може мати свій власний локальний кеш) підключиться лише до певних серверів, усі сервери Geode завжди знають одне про одного, та пов'язані одне з одним.

Системи з топологією peer-to-peer в першу чергу призначені для додатків, для яких необхідно вбудувати кеш всередині простору процесу та організуватись у кластер, рисунок 2.2. Типовим прикладом є кластер у якому додаток і кеш розташовані в одному і тому ж місці, та поділяють спільне місце.

Рисунок 2.2 – Організація топології рівноправних вузлів (peer-to-peer)

Топологія клієнт/сервер є моделлю для масштабування по-вертикалі, де клієнти, як правило, мають невелику множину даних в просторі процесу виконання та відправляють всі інші дані на сторону сервера.

У порівняння з системами, що мають топологію peer-to-peer, архітектура клієнт/сервер забезпечує набагато кращу ізоляцію даних, високу продуктивність вибірки, та кращу масштабованість. При високому навантаженні на мережу через розподіл даних, ця топологія є більш продуктивною. В будь якому варіанті топології клієнт/сервер, серверна система самостійна peer-to-peer система, з даними що розподілені між усіма серверами. Система клієнт має пул з’єднань, що використовується для зв’язку з серверами та іншими членами всередині Geode. Ще однією особливістю є те, що клієнт може містити власний локальний кеш, як показано на рисунку 2.2, що безумовно є великим плюсом.

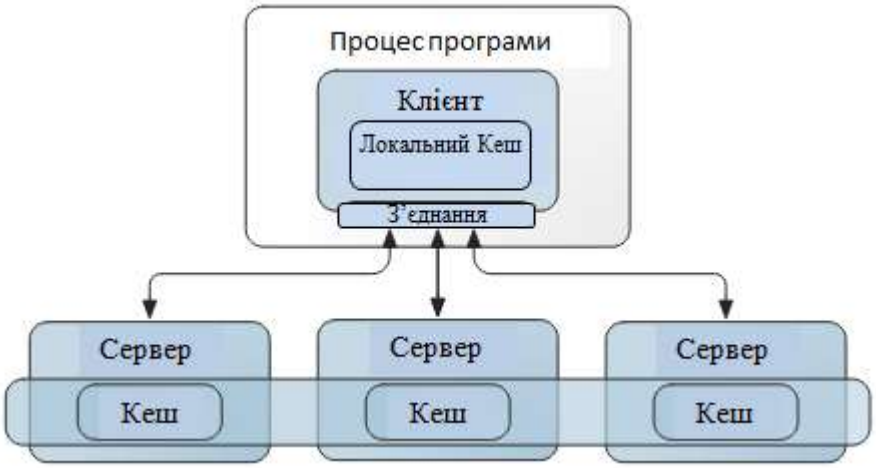

Рисунок 2.3 – Організація топології клієнт/сервер в Geode

Для масштабування по-горизонталі, є можливість використання слабо зв'язаних між собою топологію WAN, рисунок 2.4. З декількох місць, кілька Geode-систем слабо пов'язані, як правило, через географічну відстань через більш повільне з'єднання (WAN). Ця топологія забезпечує більшу продуктивність, ніж тісно зв'язана єдина система, а також дає більшу незалежність між різними місцями, що дає змогу різним вузлам функціонувати самостійно при умові, що зв'язок або віддалений сайт/ресурс з якихось причин стає недоступним. В цій топології кожен сайт є рівним одне одному, як і в топології peer-to-peer або в топології системи клієнт/сервер.

Apache Geode надає різні варіанти для взаємодії та виявлення членів як в рамках розподіленої системи, так і між клієнтами та серверами. Виявлення рівного члена (peer) – це те, що визначає розподілену систему. Всі програми і сервери кешу, що використовують одні і ті ж параметри для виявлення членів системи є членами однієї розподіленої системи. Кожен член системи має унікальний ідентифікатор та знає про ідентифікатори інших членів. Одночасно, член може належати тільки одній розподіленій системі. Після знайдення одне одного, елементи спілкуються безпосередньо одне з одним, незалежну від механізму виявлення. Для виявлення рівних членів, це розподілене сховище

даних використовує координатор для управління їх з’єднанням та відправленням [10].

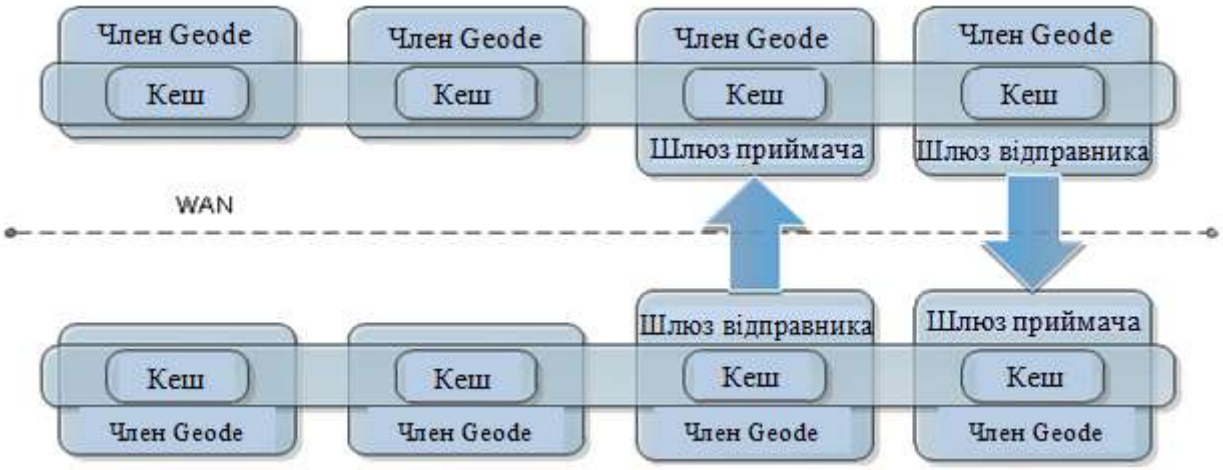

Рисунок 2.4 – Організація WAN (Multi-site) топології в Geode

Елементи системи виявляють одне одного використовуючи один, або декілька локаторів, рисунок 2.5. Локатор забезпечує як виявлення, так і балансування навантаження сервісів. Локатори управляють динамічним списком розподілених елементів системи. Усі нові члени підключаються до одного з локаторів, щоб отримати список усіх елементів, який буде використано щоб приєднатися до системи.

Слід зазначити, що кілька локаторів, замість одного, дають змогу забезпечити найстабільніший запуск та доступ для розподіленої системи.

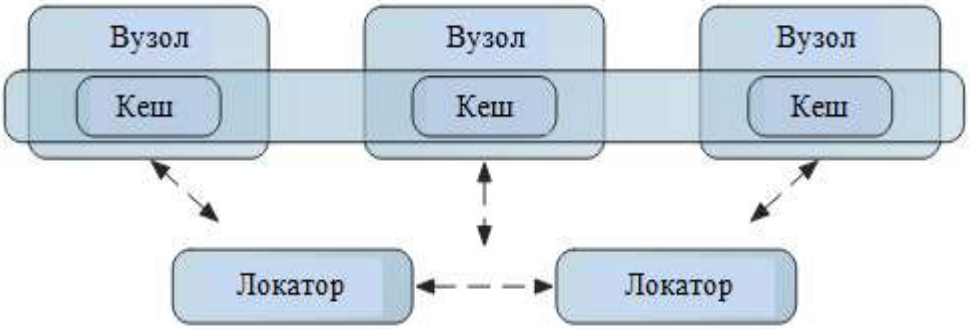

Рисунок 2.5 – Взаємодія між членами розподіленого сховища

Існують також автономні члену, що не мають аналогічних їм, а тому не використовують локаторів. Такі члени з’єднуються з розподіленою системої лише для отримання кешованих функцій. Запуск автономного члена більш швидкий та підходить для будь-якого члена, що ізольований від інших програм. Найбільше такий елемент підходить саме для клієнтських програм та додатків. До автономних елементів можна доступитися та контролювати їх за допомогою JMX менеджера.

Виявлення серверів клієнтами відбувається за допомогою локаторів, я к проілюстровано на рисунку 2.6. Локатори забезпечують клієнтів динамічним пошуком серверів та серверною пропускною здатністю. Клієнти беруть інформацію від локатора про систему серверів, та звертаються до локаторів для визначення які сервери слід використовувати. Сервери можуть змінюватись, а також їх здатність обслуговування нових клієнтських підключень або ємність може різнитися. Локтори постійно перевіряють доступність сервера та інформацію про навантаження на сервер, надаючи своїм клієнтам інформацію про ті сервери, які мають найменше навантаження [10].

Для підвищення продуктивності і когерентності кешу, клієнти повинні працювати як автономні елементи або на інших розподілених системах в порівнянні з їх серверами.

Немає необхідності запускати додаткові процеси, щоб використовувати локатори для виявлення серверів. Локатори, що забезпечують виявлення рівних членів в системі серверів також забезпечують і виявлення серверів для клієнтів. Це закладено у стандартній конфігурації Geode.

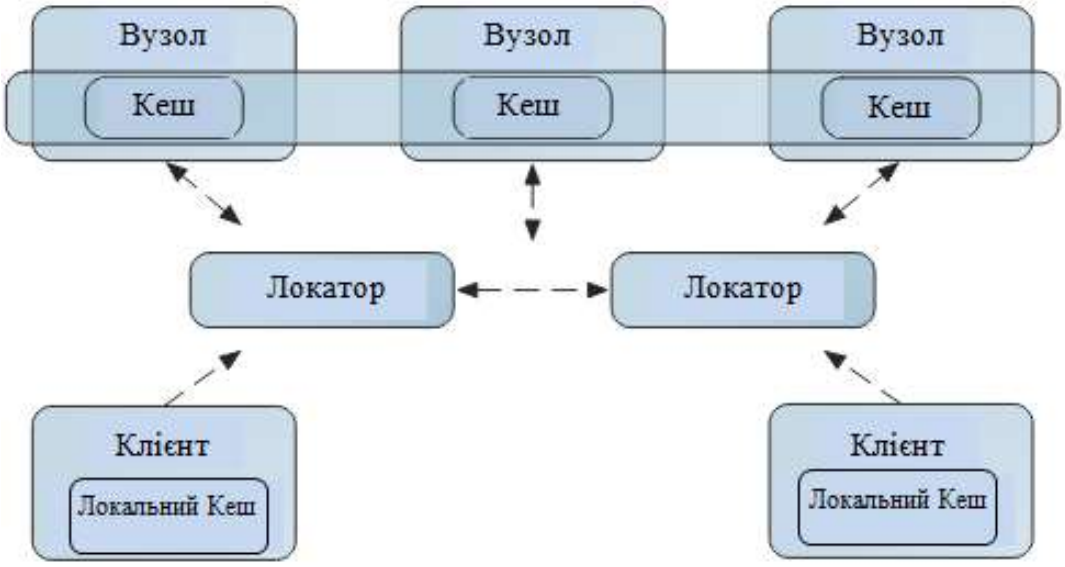

Рисунок 2.6 – Виявлення серверів клієнтами

При конфігурації системи типу WAN, кластер цього сховища використовує локатори, щоб виявити віддалені кластери, а також для виявлення локальних членів системи. Кожен локатор в такій конфігурації однозначно ідентифікує локальний кластер якому він належить, а також може ідентифікувати локатори у віддалених кластерах, до яких він буде підключатися для глобального розподілення.

Коли локатор запускається, він з'єднується з кожним віддаленим локатором для обміну інформацією про доступні локатори та шлюзи на віддаленому кластері. На додаток до обміну інформації про власний кластер, локатор передає інформацію що він отримав від інших підключених кластерів. Кожен раз, коли новий локатор запускається у системі, або існуючий локатор вимикається, зміни передаються на усі інші підключені кластери системи через WAN.

Як і всі інші In-memory-data-grid системи, Geode має свій унікальний функціонал, що виділяє його з-поміж інших розподілених сховищ даних.

Найпершим що слід виділити, це високу пропускну здатність цієї системи для зчитування та запису. Geode використовує паралельні структури даних в

основній пам’яті і високо оптимізовану розподілену інфраструктуру, для забезпечення високої пропускної здатності зчитування та запису. Програми можуть створювати копії даних у пам’яті динамічно за допомогою синхронної або асинхронної реплікації для забезпечення високої пропускної здатності зчитування, запису або розподіляти дані серед всіх членів системи для досягнення цієї цілі. Розподіл даних подвоює сукупну пропускну здатність, якщо доступ до даних збалансований по всьому набору даних. Лінійне збільшення пропускної здатності обмежено лише розміром магістральної мережі.

Однією із проблем що можуть виникнути в будь-якій IMDG системі це затримки, тому слід звертати увагу на тривалість та здатність передбачати затримки при виборі розподіленого сховища даних.

Geode мінімізує перемикання контексту між потоками та процесами. Він управляє даними в високорозвинених паралельних структурах для мінімізації точок розбрату. Зв’язок між рівноправними членами синхронний, якщо приймачі можуть працювати однаково, що дає змогу звести затримку для розподілу даних до мінімуму. Сервери керують об’єктами в серіалізованому вигляді, що дає змогу зменшити навантаження на збирач сміття.

Управління розподіленими підписками (реєстрація інтересів або безперервні запити) відбувається через сервер, що зберігає дані, гарантуючи, що підписка оброблятиметься лише один раз для всіх зацікавлених клієнтів. В результаті цього, маємо поліпшення використання процесору та використання пропускної здатності, а також зменшується час очікування для клієнтів-підписчиків.

Також слід згадати про особливості масштабування. Geode має високу масштабованість за рахунок динамічного розподілу даних між багатьма членами та розподілу навантаження даних рівномірно по всіх серверах системи. Для отримання “гарячих” даних, є можливість налаштувати систему для динамічного створення більшої кількості копій даних. Можливо також встановити поведінку

додатку, щоб він працював в розподіленій манері в безпосередній близькості від даних, що йому необхідні.

При виникненні необхідності підтримки високих та непередбачуваних сплесків паралельного навантаження клієнта, існує можливість збільшити кількість серверів, що керують даними та поширювати дані і поведінку через них, щоб забезпечити рівномірний і передбачуваний час відгуку. Клієнти постійно балансують навантаження на так званій “фермі серверів” на основі безперервного зворотного зв’язку від серверів на їх умовах навантажень. За допомогою розподілених даних між серверами, клієнти можуть динамічно переміщуватись між різними серверами для рівномірного завантаження серверів та доставки за найкращий час відгуку.

Крім того, можна поліпшити масштабованість за рахунок реалізації асинхронної здатності “писати поза” змінами даних на зовнішніх сховищах даних, як в базах даних. Geode уникає вузькі місця за допомогою запитів усіх оновлень по порядку і надмірністю. Окрім цього, є можливість порівнювати оновлення та поширювати їх в пакетному режимі в базі даних.

Як і в будь якому іншому IMDG сховищі, Geode зберігає безперервну доступність. Крім гарантованих консистентних копій даних в пам’яті, програми можуть зберігати дані на диску на один або більше членів системи синхронно або асинхронно за допомогою механізму “архітектура загальних ресурсів на диску”. Всі асинхронні події надлишково керуються принаймні двома членами, таким чином, в разі відмови одного з серверів, резервний бере керування на себе. Всі клієнти підключаються до логічних серверів, а також клієнти при збоях або через недоступність серверу автоматично переключаються на альтернативні сервери в групі.

Слід сказати також про механізм розкладу повідомлень про події. Системи публікацій та опису пропонують послугу поширення даних, де нові події публікуються до системи та надсилаються до усіх зацікавлених підписчиків

надійним способом. Традиційні платформи обміну повідомленнями зосереджені на доставці повідомлень, але часто додатки-приймачі потребують доступу до пов’язаних з ними даних до того, як вони можуть обробити подію. Це вимагає від них доступу до бази даних, коли ця подія доставляється, обмежуючи швидкість підписчика швидкістю бази даних.

Geode пропонує дані та події через єдину систему. Дані управляються як об’єкти в один або більше розподілених регіонів, аналогічних до таблиць в базах даних. Програмі достатньо просто додати, оновити або видалити об’єкт в регіоні і платформа повідомляє про ці зміни усіх підписчиків. Підписчик, що приймає подію має прямий доступ до спільних даних в локальній пам’яті або може отримати дані з одного або з іншого члена за один стрибок.

Паралельне виконання програми в IMDG системах є однією з ключових функцій. Geode підтримує виконання бізнес-логіки програми паралельно на різних членах сховища. Різні сервіси цього сховища допускають виконання довільних, залежних від даних функцій на членах системи, де дані розподілені на локальні частини за відліком та масштабом.

За допомогою розміщення релевантних даних та розпаралелив обчислення, можна збільшити загальну пропускну здатність. Розрахунок часу очікування обернено пропорційний кількості членів, на яких він може бути розпаралелений.

Основною передумовою є передача функції прозоро для програми, що несе у собі множину даних, що необхідні цій функції, а також щоб уникнути переміщення даних поза мережею. Функція програми може виконуватись лише на одному члені, паралельно на множині членів, або ж паралельно на усіх членах системи. Ця програмна модель схожа на популярно модель Map-Reduce, що була запропонована компанією Google. Маршрутизація проінформованої даними (Data-aware) функції найбільше підходить для програм, що потребують ітерацій по безлічі елементам даних (наприклад, запит або визначена функція агрегації).

Повертаючись до архітектури сталих загальних ресурсів на диску, слід сказати що кожен учасник системи керує даними на диску незалежно від інших членів. Неполадки диску або збої в кеші одного члена ніяк не впливає на здатність іншого безпечно працювати на своїх файлах диску. Ця “живуча” архітектура дозволяє програмам бути організованими таким чином, що різні класи даних зберігаються на різних членах системи, що значно збільшує пропускну здатність програми, навіть за умови що диск налаштований лише на стійкість для об’єктів програми [8].

На відміну від традиційної системи бази даних, Geode не керує даними і транзакційними журналами в окремих файлах. Усі оновлення даних додаються до файлів, що схожі на традиційні логи транзакцій в базах даних. Таким чином можна уникнути час, під час якого були певні проблеми з диском, якщо диск не використовувався одночасно іншими процесами, а отже витрати йдуть лише на ротаційні затримки.

Ще однією суттєвою особливістю є можливість клієнтів передавати окремі запити безпосередньо до серверу, що містить ключ даних, уникаючи декількох переходів для пошуку даних, що розбиті на частини. Метадані клієнту дають змогу ідентифікувати правильний сервер. Таким чином, ця функція суттєво підвищує продуктивність та швидкодію клієнтського доступу до розподілених регіонів на рівні серверу.

Що стосується безпеки, то в цьому розподіленому сховищі є механізми для її забезпечення. Підтримується запуск декількох окремих користувачів в клієнтських програмах. Ця можливість організована таким чином, що клієнти сховища вбудовані в сервери програми, і кожен сервер підтримує запити від багатьох користувачів. Кожен користувач може отримати доступ до невеликої множени даних на серверах, так як в клієнтській програмі, де кожен користувач може отримати доступ лише до своїх власних замовлень. Кожен користувач

клієнта підключається до сервера з власним набором облікових даних і має власний дозвіл для доступу до кешу.

При використанні топології клієнт/сервер зв’язок має більш високий рівень захисту від атак. Сервер відправляє клієнту унікальний, випадковий ідентифікатор разом з кожної відповіддю, що буде використаний при наступному клієнтському запиті. Через ідентифікатор, навіть повторний виклик операції клієнту надсилається в якості унікального запиту до серверу.

Повертаючись до багатовузлового розподілу даних, проблеми масштабованості можуть виникати в результаті сайтів даних, що поширюються географічно через глобальну мережу (WAN). Geode пропонує модель для вирішення проблем таких топологіях, починаючи від одного кластера типу peer-to-peer до надійного зв’язку між центрами обробки даних через WAN. Ця модель дозволяє масштабувати розподілені системи необмежено та надає можливість використання слабкого зв’язку без втрат продуктивності, надійності або узгодженості даних.

В основі цієї архітектури лежить конфігурація шлюзу відправника, що використовується для розподілення регіональних подій до віддаленого сайту. Надається змога розгортання екземпляру шлюзу відправника паралельно, що дозволяє сховищу збільшити пропускну здатність для розподілених регіональних подій через WAN. Можна також налаштувати шлюз черги відправника для забезпечення високої доступності та довготривалого зберігання щоб уникнути втрати даних у разі виходу з ладу одного з членів системи.

Наостанок слід сказати, що Geode підтримує тривалі запити. У сучасних системах обміну повідомленнями накшталт Java Message Service, клієнти підписуються на отримання повідомлень за конкретними темами. Будь-яке повідомлення, що доставляється по конкретній темі пересилається підписчику. Можливість обробки тривалих запитів можна організовувати за допомогою програм, використовуючи Object Query Language (OQL) [8].

2.2 Розробка інструменту динамічного завантаження класів

2.2.1 Алгоритм інструменту динамічного завантаження

Алгоритм динамічного завантаження було організовано як позначено на рисунку 2.6.

Новий клас для завантаження надходить до IMDG зі сторони клієнта разом с додатковою інформацією, необхідною для генерації конвертору.

Слід наголосити, що створені об’єкти нового класу не заміщують старі, доки не буде відтворено всі старі об’єкти. Таке рішення може потребувати великої кількості пам’яті. У випадку нестачі оперативної пам’яті можливо зберігати отримані об’єкти на жорсткому диску, що значно сповільнить процес. Але таке рішення забезпечує вирішення одразу двох проблем.

Перша – це транзакційність процесу. Якщо при конвертації виникне помилка несумістності полів старого та нового класу – про це буде повідомлений клієнт і зміни вже вдало перетворених об’єктів внесені не будуть, що дозволить запобігти змішанню старих та нових об’єктів. В такому випадку клієнт зможе надіслати іншу версію оновленого класу та повторити спробу.

Друга – це проблема доступу під час перезавантаження. Адже весь процес займає час, в який, при іншому алгоритмі, доступ до сховища був би неможливим. Завдяки обраній стратегії, на етапі перестворення об’єктів, неможливе буде лише змінення та видалення об’єктів зі сховища. Під час заміщення запити на отримання об’єктів класу, що оновлюються, будуть задовольнятися за рахунок перестворених об’єктів, а додавання нових об’єктів, у випадку коли вони реалізують старий клас, проходить через конвертор. Таким чином вже на етапі заміщення сховище буде оперувати об’єктами нового класу.

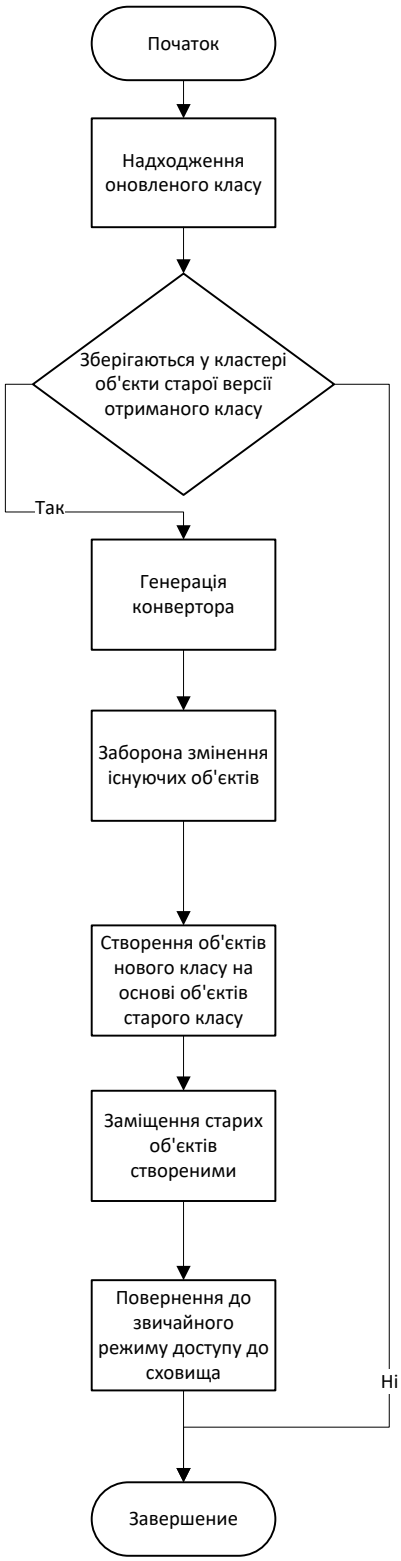

Рисунок 2.6 – Алгоритм роботи інструменту динамічного завантаження

2.2.2. Вибір інструментів розробки програми

В наш час є величезний вибір різних інструментів для розробки програмних засобів.

Найпопулярнішими мовами програмування зараз є: Java, C#, C++ та інші. Кожна з існуючих мов має свої переваги та недоліки при вирішенні тієї чи іншої задачі. Через такі чинники як: швидкість, надійність, кросплатформеність та простота використання автором була обрана саме мова програмування Java. Java також є однією з найбільш поширених мов для вирішення різних задач, що пов'язані з розподіленими сховищами даних.

Окрім мови програмування, необхідно було визначитись з середовищем розробки. На даний момент найбільш популярними середовищами розробки на мові програмування Java, є: Intellij Idea, Eclipse та NetBeans.

Intellij Idea – комерційне інтегроване середовище розробки на різних мовах програмування від компанії JetBrains. Існує безкоштовна версія цього середовища розробки "Community Edition" . Community версія підтримує інструменти для проведення тестування у вигляді TestNG та JUnit, системи контролю версій CSV, Subversion, Mercurial та Git, засоби складання Maven, Ant, Gradle, мови програмування Java, Scala, Cloujure, Groovy та інші. Підтримується розробка мобільних додатків, для мобільної платформи Android. До складу входить модуль візуального програмування GUI-інтерфейсу Swing UI Designer, XML-редактор, редактор регулярних виразів, система перевірки коректності коду, система контролю за виконанням завдань і доповнення для імпорту та експорту проектів з інших середовищ розробки. Доступні засоби інтеграції з системами відстеження помилок Jira, Trac, Redmine, Pivotal Tracker, GitHub та іншими.

Eclipse – вільне модульне інтегроване середовище розробки програмного забезпечення. Розробляється і підтримується Eclipse Foundation і включає проекти, такі як платформа Eclipse, набір інструментів для розробників на мові Java, засоби для управління сирцевими кодами, візуальні побудовники GUI

тощо. Написаний в основному на Java, може бути використаний для розробки на Java, а за допомогою різних плагінів і на інших мовах програмування, включаючи Ada, C, C++. COBOL. Perl, PHP та інші. Підтримує велику кількість популярних нині технологій. Взаємодіє із системами контролю версій. Різні розширення дають змогу використання нових технологій накшталт Spring, як інтегрованого інструменту.

Net Beans – вільне інтегроване середовище розробки для мов програмування Java, JavaFX, C, C++, PHP, тощо. Середовище може бути встановлене і для підтримки окремих мов, і у повній конфігурації. Поширюється у сирцевих текстах під ліцензіями GPLv2 і CDDL. Проект NetBeans IDE підтримувався і спонсорувався фірмою Sun Microsystems, нині Oracle. За якістю і можливостями останні версії замагаються з найкращими інтегрованими середовищами розробки для мови Java, підтримуючи рефакторинг, профілювання, виділення синтаксичних конструкцій кольором, автодоповнення мовних конструкцій на льоту, шаблони коду та інше. Net Beans підтримує плагіни, дозволяючи розробникам розширювати можливості середовища. Також підтримуються різні сучасні технології [16].

Розглянувши переваги і недоліки різних середовищ розробки, було вирішено обрати Intellij Idea, вперш за все через здатність цього середовища до автоматичного корегування, та великому спрощенню написання об’ємного коду завдяки сучасному і зручному графічному інтерфейсу.

2.2.3. Аналіз ефективності динамічного завантаження

У використаній моделі розподіленого сховища даних завантаження класів відбувається при старті елементу системи за стандартною схемою завантаження класів на сервер. При старті, JVM використовує стандартний завантажник класів bootstrap classloader для завантаження базових необхідних класів, далі через системний завантажник відбувається завантаження необхідних класів, шлях до яких зберігається в змінних середовища, та може бути заданий за допомогою

команди через консоль. На рисунку 2.7. відображено діаграму послідовності завантаження класів.

Рисунок 2.7. Діаграма послідовності завантаження класів.

Як би то не було, користувач системи повинен запаковувати усі класи, що мають бути використані у розподіленому сховищі даних до jar архівів, а далі розгортати за допомогою консолі. Це призводить до необхідності перезавантаження системи, або окремої частини, де і буде використаний

необхідний клас, і, відповідно, при необхідності внесення певних змін, неможливо буде це зробити без втрати даних, а також такий підхід потребує часу на перезапуск.

Підхід динамічного завантаження дає змогу уникнути цих проблем. Перезавантаження будь-якого класу може бути зиніційовано в будь-який момент часу, за умови що в цей час вже не відбувається перезавантаження цього класу. Під час операції завантаження тимчасово блокується можливість змінювати вже існуючі об’єкти старої версії класу, коли у випадку перезавантаження кластера доступ був би втрачений повністю.

ВИСНОВКИ ДО РОЗДІЛУ 2

Існує досить великий набір рішень IMDG, створених на різних мовах програмування. Проте, жодна з існуючих систем не реалізує динамічне завантаження класів, Що зумовило актуальність розробки інструменту для вирішення цього питання.

Розгляд актуальних мов програмування та середовища розробки задля вибору найбільш зручної для вирішення обраної проблеми дозволив виокремити мову java та середовище розробки intellij IDEA. Виходячи із нагальних потреб, а саме доступності та відкритості сирцевого коду, було вирішено використовувати для необхідних цілей розподілене сховище даних, що не є комерційним, розроблене компанією Apache – Geode.

Детальний розгляд можливостей та внутрішнього устрою Apache Geode, а також механізмів завантаження класів дозволив розробити алгоритм роботи інструменту динамічного завантаження класів зі збереженням даних. Також, в межах проведеного дослідження, були визначені особливості застосування алгоритму роботи інструменту динамічного завантаження класів в контексті

використання у розподіленому сховищі даних в оперативній пам’яті з урахуванням можливої нестачі доступного об’єму пам’яті.

Було встановлено, що динамічне завантаження класів дає змогу уникнути проблем, які виникають при перезавантаження системи, або окремої частини.

РОЗДІЛ 3. Реалізація моделі інструменту динамічного завантаження

3.1. Модель інструменту динамічного завантаження

Ідея динамічного завантаження класів, як і ідея звичайного завантаження, полягає у тому, що коли певні класи зазнали змін, що впливають на виконання програми, то необхідно перекомпілювати саме змінений файл для отримання актуальних результатів після проведених змін. Відмінністю є той факт, що при звичайному завантаженні є можливість призупинити сервіс або програму, що залежить від зміненого класу, перекомпілювати необхідний ресурс, а згодом запустити всю систему з нуля, або перезапустити лише ту частину, на яку вплинули ці зміни. Під час перезапуску можлива втрата певних вже існуючих даних, не кажучи про збільшення часу розробки, який витрачається на кожний перезапуск у системі. Динамічне завантаження ж дає змогу спростити та пришвидшити розробку шляхом завантаження класів без перезапуску системи, в режимі реального часу.

Слід зазначити, що при різних умовах зміненості в класі змінюється і складність його перезавантаження, що необхідно враховувати під час створення системи. Існує три основні варіанти змін:

1) Зміні імені поля;
2) Зміні типу поля;
3) Зміна поля у вкладеному об’єкті.

Щодо першого варіанту, то він є найпростішим з точки зору технічної реалізації. Достатньо лише переіменувати дане поле, і оповістити про цю зміну систему загалом.

Під час зміненого типу поля, окрім самої зміни та оповіщення, існує необхідність привести тип, і прослідкувати за тим, що б система також змогла працювати з новим по наповненню об’єктом.

Останній варіант, а саме за умови що змінюється вкладений об’єкт, мало

чим відрізняється від попередніх двох за алгоритмом змін, проте виникає проблема складності роботи з цим самим об’єктом. В даному випадку набагато складніше скласти мапу залежностей, що в подальшому буде відповідати за відповідність змін пов’язаних з даним об’єктом для всієї системи.

Також в описаному варіанті можуть виникнути складнощі саме зі швидкодією роботи системи. Така проблем пов’язана з тим, що працювати з об’єктом напряму дозволяє простий доступ до нього, а із вкладеним необхідно звертатися стільки разів, у скільки об’єкт вкладено у основний клас.

Для забезпечення простоти та можливості використання інструменту динамічного завантаження в різних сервісах, було вирішено розробити модель, що повинна визначати зміни у файлах, що задіяні, а також самостійно завантажувати змінені класи. Маючи на увазі те, що перезавантаження має відбуватись в IMDG, було вирішено позиціонувати програму як доповнення до розподіленого сховища даних, що починає перезавантаження при надсиланні клієнтом класу. Так як за основу було взято мову програмування Java, то, звичайно ж, є можливість використання різних вже існуючих рішень для вирішення тих, чи інших задач.

3.2 Реалізація інструменту динамічного завантаження

Алгоритм реалізації інструменту динамічного завантаження було організовано як позначено на рисунку 3.1.

Отже, завантаження проходить в 4 етапи:

1) Ініціалізація: завантаження класу, конверторів, створення мапи зав’язків полів та конвертерів на основі описуючих фалів;

2) Підготовка: генерація конвертерів;

3) Обробка: створення нових об’єктів на основі старих;

4) Заміщення: заміщення старих даних новими.

Зосередимо увагу на перших двох етапах.

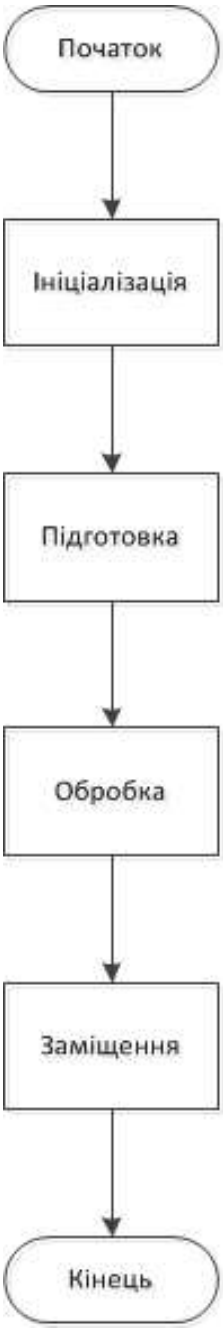

Рисунок 3.1 – Алгоритм реалізації інструменту динамічного завантаження

3.3 Реалізація завантажника класів

Завантажник класів відповідає за внесення до програми надісланого класу. В даному випадку, класом завантажником є клас DynamicClassOverloader, рисунок 3.2. На відміну від стандартного завантажника він здатен завантажити клас навіть якщо клас с таким ім'ям вже існує у програмі. Саме він використовується для завантаження отриманого від клієнта класу. Також він використовується для завантаження сторонніх конверторів.

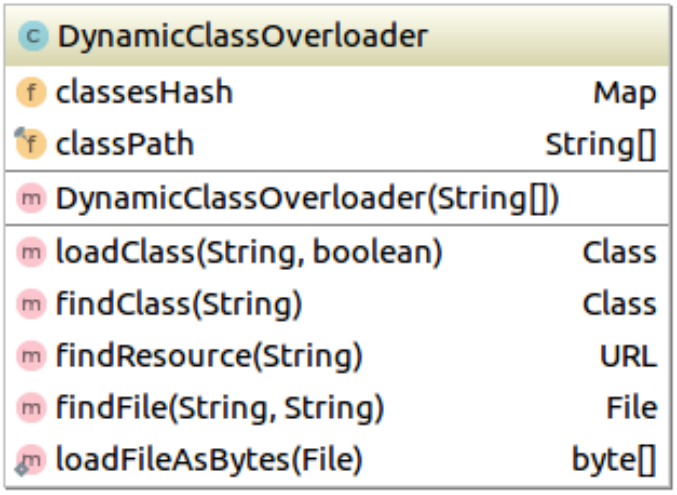

Рисунок 3.2 – Реалізований клас DynamicClassOverloader

Після завантаження класу та конверторів відбувається створення мап відповідностей на основі наданих клієнтом описуючих файлів.

3.4 Механізм конвертації полів з різним типом

Перетворення об'єктів старого класу на об'єкти нового відбувається за рахунок рефлексії. За своєю сутністю, рефлексія – це механізм дослідження даних про програму під час її виконання. Саме за допомогою цього інструменту є можливість досліджувати інформацію щодо полів, методів та конструкторів

класів. Також, цей механізм надає можливість безпосередньо виконувати операції над полями або методами. Однією із можливостей рефлексії є можливість почергового переносу значень полів старого об’єкту до відповідних полів нового об’єкту з урахуванням зміни типу полів, що і забезпечує розроблений клас SmartCaster, рисунок 3.3.

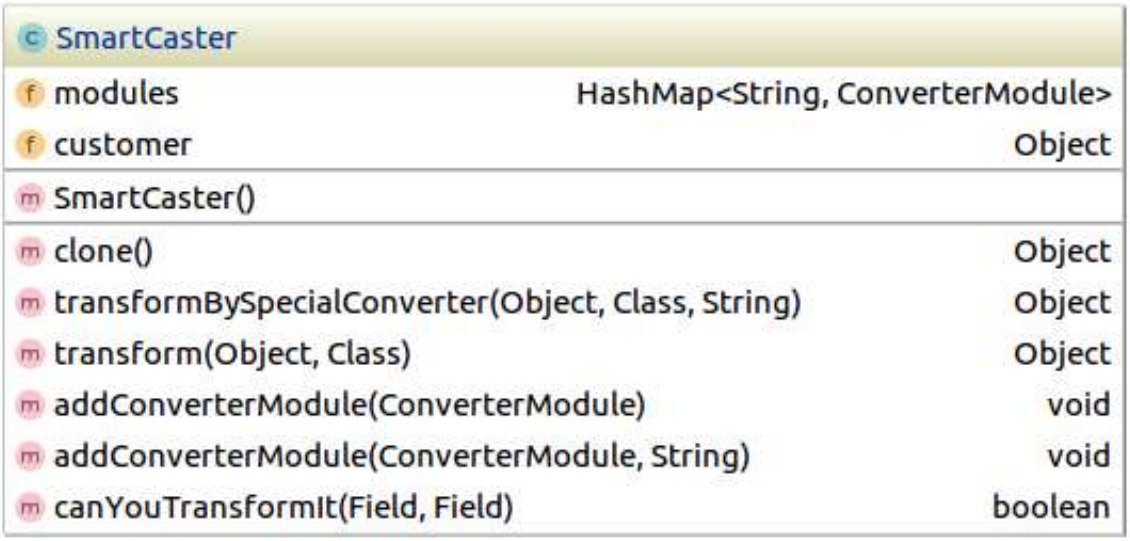

Рисунок 3.3 Діаграма класу SmartCaster

Для цього було створено метод що приймають Object, який необхідно конвертувати та Class, об’єктом якого має бути результат. Звісно, задля конвертації необхідно мати інструкції як само має відбуватись конвертація. Такі інструкції надаються SmartCaster за допомогою об’єктів класу, що реалізують інтерфейс ConverterModule.

Откримати такі модулі SmartCaster може двома шляхами. Перший шлях – напряму від клієнта. Разом із класом, що необхідно завантажити, клієнт може відправити набір конверторів, що необхідні для переносу даних із об’єктів старого класу до нового. Таким чином якщо клієнтові необхідно якось перетворювати дані при переносі – він має можливість це зробити так, як вважає необхідним, достатньо лише створити та прикласти відповідний клас. За бажання можна створити окремий конвертор для окремих полів, що реалізується за

допомогою додаткових файлів, що описують зв'язки між полями та конверторами. Такий підхід робить систему більш гнучкою та дає більшу свободу клієнту. Наприклад введення "порожнього" конвертора дозволяє ігнорувати зв'язки між полями, якщо в цьому э потреба, або можна створити конвертор, що заповнює поле значенням за замовчанням, якщо це значення відмінне від значення за замовчанням для об'єктів, що будуть додані після завантаження.

Другий шлях – генерація власного конвертора за потребою. Це відбувається у тих випадках, коли відповідні поля мають різні не примітивні типи та жоден з модулів не може задовільнити потребу.

3.5 Генерація конвертору

Найскладнішим етапом розробки є саме генерація конвертору. Для цього будо створено клас GeneratedConverter, рисунок 3.4. Суть генерації в створенні мапи відповідності полів старого та нового класу та забезпечення наявності відповідних модулів-конвертерів задля забезпечення вдалого переносу даних.

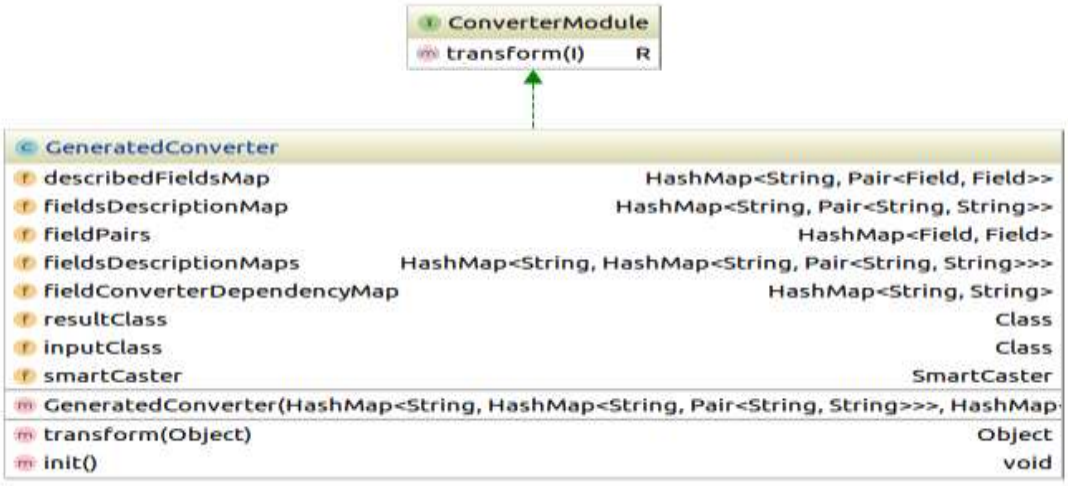

Рисунок 3.4 – Реалізований клас GeneratedConverter

Відповідність між полями визначається двома шляхами. Перший це

знаходження полів з відповідними іменами, для тих випадків, коли при зміні класу поля не зазнали змін чи змінені будуть лише типи даних. Другий за прикладеним клієнтом файлом з описом зв’язків полів в старому та новому класі. Таким чином, клієнт може забезпечити коректне перенесення даних у випадку зміни назви поля, або в будь-якому іншому випадку, коли назви для створення зв’язку недостатньо.

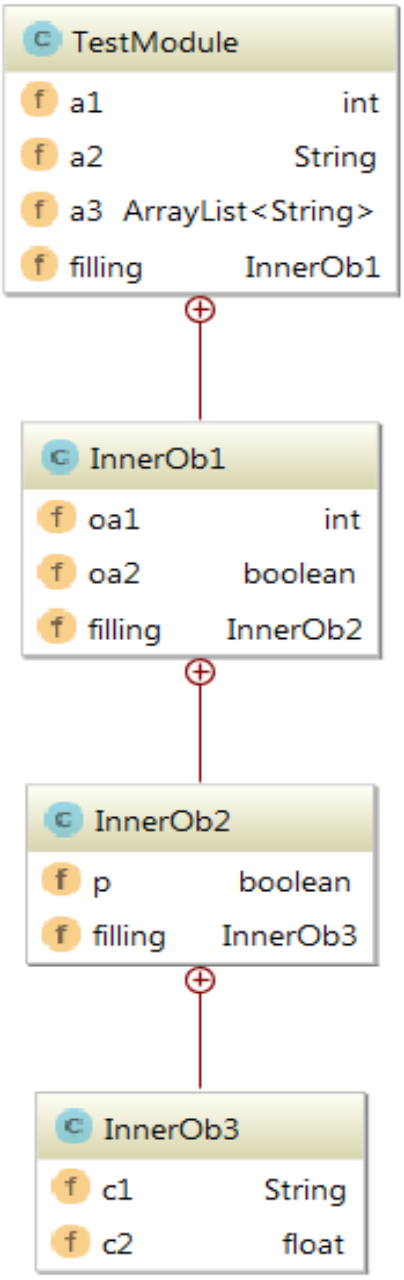

Рисунок 3.5 Тестовий клас TestModule

Після створення мапи зав’язків полів відбувається перевірка наявності

конверторів для кожного з полів. У випадку, якщо необхідного конвертора у SmartCaster немає а поля не є примітивними генерується новий конвертор. Таким чином реалізується рекурсія. Рекурсія також забезпечує обробку вкладених класів. Тестування проводилось на класі з 3 вкладеними один в одне класами, рисунок 3.5.

3.6. Застосування механізму завантаження в розподіленому сховищі

Інструмент позиціонується як додаткове програмне забезпечення для IMDG, що дає перевагу до швидкодії виконання перезавантаження зрівнюючи із варіантом цілком стороннього програмного забезпечення зі сторони клієнту, адже це дає змогу виконувати перезавантаження розподілено, бо для кожного кластеру перетворення будуть проходити окремо. Задля ще більшого прискорення процесу було розроблено алгоритм паралельного виконання этапу обробки.

Необхідно зазначити, що нажаль зробити повністю автоматизовану систему не вдалося. Це пов’язано саме із необхідністю надання системі інструкцій зі сторони клієнта. Звичайно існує можливість використання вже існуючих сервісів спостереження за змінами об’єктів всередині системи, про те вони доволі сильно збільшують сумарний час роботи системи, і не завжди правильно розпізнають абсолютно усі зміни, що є недопустимим для коректної роботи клієнтських сервісів.

Слід детальніше зупинитися на етапі заміщення. В другому розділі було зазначено, що під час етапу обробки нові об’єкти до етапу заміщення зберігаються окремо, що робить завантаження безпечним та звільняє від необхідності зупинення доступу до даних під час виконання. Це є ідеальним варіантом, доки на це вистачає пам’яті. Але окрім збереження на жорсткому

диску замість оперативної пам’яті слід розглянути ще один варіант.

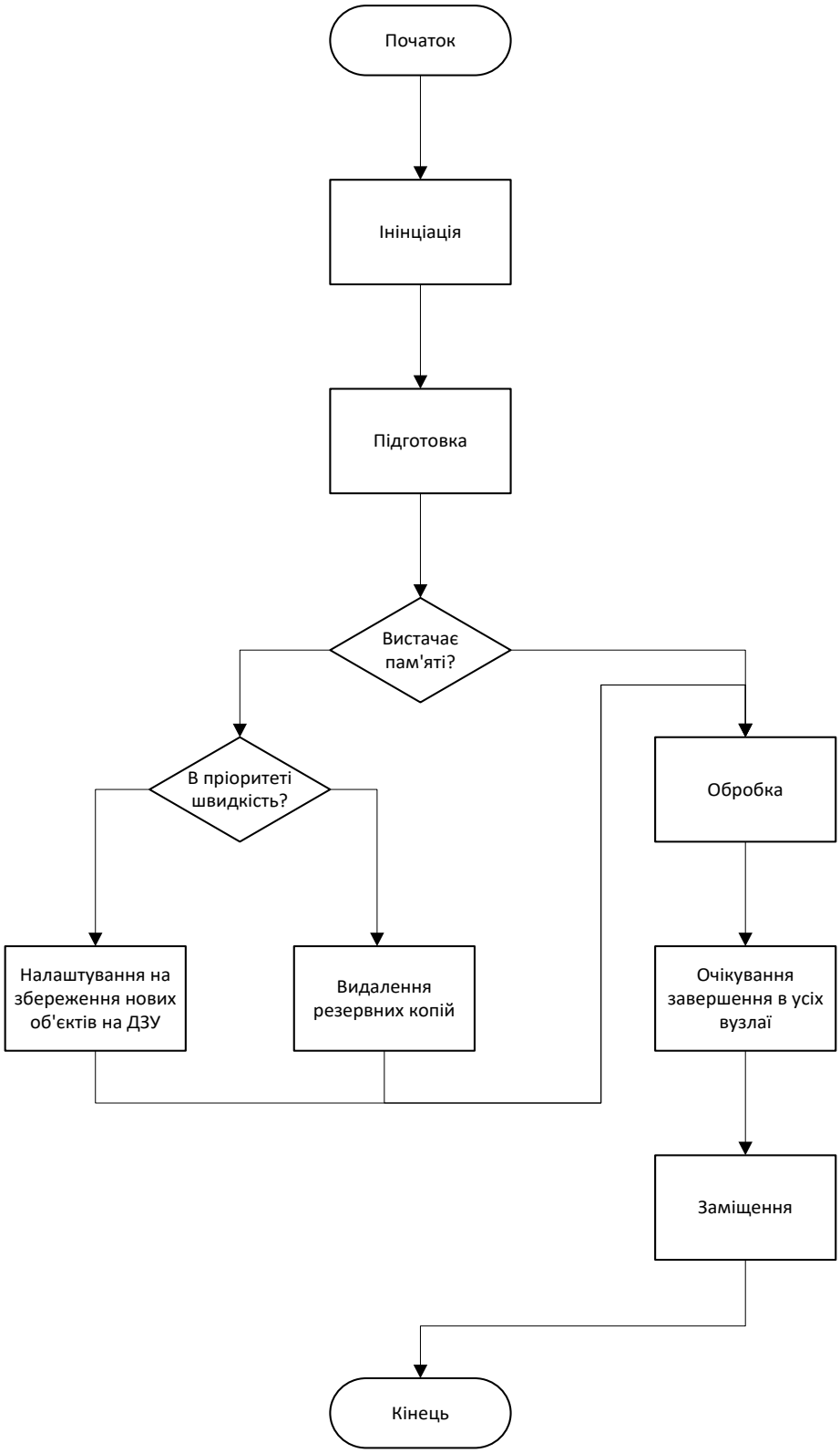

Рисунок 3.6 – Блок-схема алгоритму дій у контексті IMDG

Задля забезпечення безпеки на випадок втрати даних одного з кластерів

інші кластери в деяких IMDG дублюють інформацію одне одного. Таким чином дані можливо буде відновити. Якщо використовувати розроблений інструмент в розподіленому сховищі з такою системою дублювання у випадку, коли затримки в часі є критичними, можливим варіантом є видалення дубльованих даних задля звільнення пам’яті, що призведе до тимчасової зниження надійності сховища, але значно прискорить процес у порівнянні з варіантом використання пам’яті жорсткого диску. Таким чином алгоритм дій набуває вигляду, зазначеного на рисунку 3.6.

3.7. Інструкція користувачеві

Розроблена програма імітує спрощену роботу вузла. Вона зберігає в собі набір об’єктів класу EzTest, рисунок 3.7. При запуску вузол не містить жодного об’єкту. На рисунку 3.7 зображена діаграма класу EzTest.

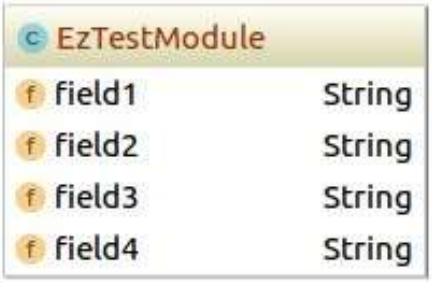

Рисунок 3.7 – Діаграма класу EzTest

Користувачеві доступний ряд консольних команд для взаємодії:

– more - Команда створює у вузлі n об’єктів класу EzTest та виводить наявну кількість об’єктів, де n наступне ціле додатнє число, введене користувачем;
– clear – видаляє усі наявні об’єктів та оповіщає про вдале завершення очищення;

– setClass classPath fieldsDescriptionPaths fieldConverterDependencyPaths converterModulesPath converterModulesName1 … converterModulesNameN – завантажує до програми новий файл з розширенням .class.

Параметри після команди:

– classPath – шлях до класу;
– fieldsDescriptionPaths – шлях до файлу, що описує зв’язки між полями;
– fieldConverterDependencyPaths – шлях до файлу, що описує зв’язки між парами полів та конвертером;
– converterModulesPath – шлях до директорії, що містить конвертори.

Всі наступні параметри будуть розглядатися як імена конверторів, що мають бути завантажені.

reload – ініціює процес завантаження класу та оновлення всіх наявних об’єктів. Під час виконання буде виводитися прогрес у вигляді знаків «!». Кожен виведений знак свідчить про завершення 10% всього обсягу наявних класів. Після завершення буде виведен час, що було витрачено на перезавантаження.

reloadp – ініціює процес завантаження класу та оновлення всіх наявних об’єктів на основі багато поточного алгоритму. Прогрес також буде виводитися, але замість знаку «!» буде виводитися номер ядра (процесора) що закінчив виконання цього проценту задач.

У разі, коли reload або reloadp буди выкликаны до виклику команди setClass перезавантаження буде виконано на основі класу по замовченню.

exit – завершення виконання програми.

На рисунку 3.8 зображено вивід консолі при вдалому перезавантаженні класу на основі класів за замовчанням (що зберігаються у директоріях classHolder\pocket1 та classHolder\pocket2).

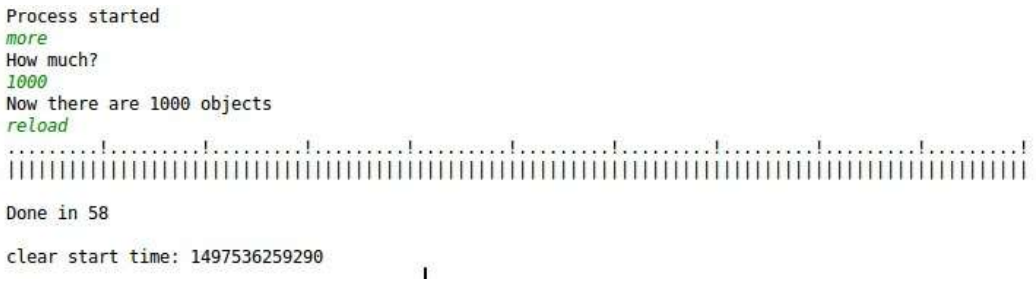

Рисунок 3.8 – Вивід консолі при вдалому перезавантаженні класу на основі класів за замовчуванням

ВИСНОВКИ ДО РОЗДІЛУ 3

В даному розділі було описано реалізацію алгоритму динамічного завантаження класів, в рамках чого була представлена концепція використання генерації конверторів з можливістю рекурсивного виклику задля забезпечення збереження інформації при динамічному перезавантаженні класів, що дозволяє коректно переносити інформацію, навіть у випадках високої вкладеності з мінімальною кількістю супроводжуючих інструкцій.

Зазначено, що повністю автоматизовану систему розробити не вдалося. Це пов’язано саме із необхідністю надання системі інструкцій зі сторони клієнта. Звичайно існує можливість використання вже існуючих сервісів спостереження за змінами об’єктів всередині системи, про те вони доволі сильно збільшують сумарний час роботи системи, і не завжди правильно розпізнають абсолютно усі зміни, що є недопустимим для коректної роботи клієнтських сервісів.

Було наголошено на необхідних умовах використання алгоритму на боці IMDG та наведено інструкцію користування програмою. Продемонстровано працездатність запропонованої системи вирішення проблеми динамічного завантаження у розподіленому сховищі даних в оперативній пам’яті.

РОЗДІЛ 4. Впровадження системи динамічного завантаження класів у розподілених сховищах даних

4.1 Загальний огляд тестування системи та критерії тестування.

Тестування є одним з найбільш необхідних інструментів для визначення відповідності розробленої системи до цілей, заради яких вона була створена. Для будь-якої системи існує безліч критеріїв, за якими визначається її майбутній попит у використанні. Узагальнюючи характеристики системи, слід виділити основні три, що мають найбільш вагоме значення при її роботі, а саме: швидкість, надійність та ресурсоємність.

В даній системі ресурсоємність більшою мірою залежить саме від розподіленого сховища даних в оперативній пам’яті та наявних ресурсів безпосередньо у ньому. Сам механізм динамічного завантаження не потребує надвисокого використання ресурсів, так як складність виконання будь-яких дій цілком залежить саме від розміру змінюваного об’єкту та кількості його полів. В свою чергу саме розміри необхідних класів, а також кількість їх заміщених копій, що саме і зберігаються в IMDG, потребують найбільше ресурсів, що використовує дана система.

Саме швидкість та надійність є критично необхідними критеріями для створеною системи. Як було зазначено, основною ціллю системи є прискорення розробки, та звісно ж покращення показників роботи сервісів за рахунок відмови від перезавантаження IMDG.

Розроблена система має бути застосована в широкому спектрі сервісів. На сучасному ринку, однією з головних умов успішності будь-якого сервісу є доступність в будь-який час, незважаючи на функціонал, що не буде мати жодного сенсу, якщо сервіс не буде стабільним. Саме тому, розроблена система має працювати стабільно та виконувати покладені на неї функції.

Отже, було вирішено провести низку тестів що покажуть швидкодію

розробленої системи, а також реакцію системи на різну кількість навантаження, за умови однакових фізичних ресурсів.

Заради найбільш об’єктивної оцінки було вирішено тестувати розроблену систему при трьох різних умовах.

В першому варіанті, тестування буде проводитись за умови, що ім’я одного з полів зміненого об’єкту буде відрізнятися. Відповідно система повинна провести зміни таким чином, щоб і решта членів сервісу змогла використовувати те ж поле, але тепер з іншим ім’ям. Таким чином буде здійснено перевірку надійності даної системи.

У другому варіанті, суть тестів криється у тому, що ім’я поля зміненого об’єкту залишається сталим, про те тип цього поля буде змінено. Таким чином, системі необхідно, без втрати даних, виконати маніпуляції що призведуть до безболісної конвертації поля цього об’єкта, а також система не зазнає втрат або шкоди.

Третій варіант відрізняється суттєво та є більш складним, в даному варіанті тестування у змінюваному об’єкті буде змінюватись як назва поля, так і його тип. Про те, ці маніпуляції будуть відбуватися на об’єкті, що є частиною змінюваного. За рахунок такого підходу, є можливість перевірити швидкодію системи, а також за допомогою маніпуляцій саме на об’єкті, всередині змінюваного – те, наскільки система працює надійно, і чи не відбуватимуться певні втрати даних, як всередині даного об’єкта, так і на всьому запущеному сервісі, що від нього залежить.

Завдяки різноманітному підходу до аналізу системи, є можливість оцінити її з різних ракурсів. Зрозуміти, чи є даний підхід до вирішення проблеми динамічного завантаження в розподілених сховищах даних в оперативній пам’яті актуальним для всіх сервісів, що працюють на базі IMDG, чи все ж існують певні обмеження, пов’язані з різними факторами, що можуть вплинути на працездатність сервісів.

Для проведення тестування було створено спрощений клас TestModule який містить в собі InnerObject, рисунок 4.1.

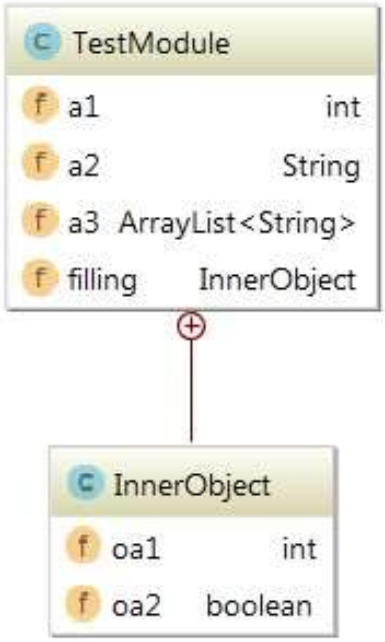

Рисунок 4.1 – Діаграма класу TestModule

За замовчуванням, системі необхідно надати інструкцію, що забезпечить збереження всіх попередніх версій файлу, або ж навпаки після вдалого перезавантаження класу і імплементації його до робочого сервісу – видалить усі не актуальні копії. Через те, що тестування відбувається у не комерційній площині і ресурси обмежуються 4 гігабайтами ОЗП, потрібно проаналізувати наскільки доцільним буде збереження попередніх версій всередині системи.

Було вирішено проводити тестування у три етапи для будь-якого з критеріїв:

1) Система створює та обробляє 10 об’єктів, із зберіганням попередніх версій об’єкту;

2) Система створює та обробляє 100 об’єктів, із зберіганням попередніх версій об’єкту;

3) Система створює та обробляє 1000 об’єктів, з подальшим

видаленням усіх не актуальних версій об’єкту.

Таким чином буде забезпечена повноцінна перевірка працездатності системи відповідно до наявного технічного обладнання.

Тестування при зміні імені поля. Відповідно до першого варіанту тестування, в змінюваному об’єкті змінюється лише назва одного з полів, рисунок 4.2.

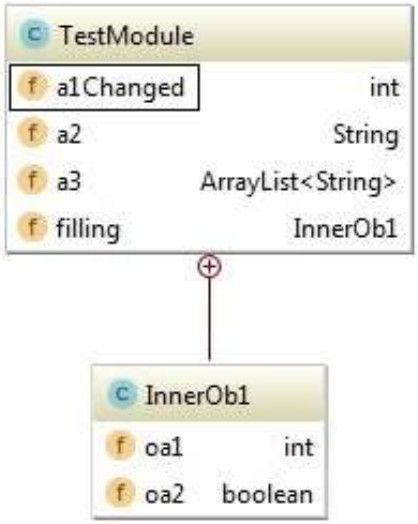

Рисунок 4.2 – Клас зі зміненим ім’ям поля

У відповідності до алгоритму виконання, програма згідно інструкцій надісланих клієнтом отримує дані, про те, що клас змінено. Після чого починається ініціалізація системи динамічного завантаження класів у розподіленому сховищі. У тестових випадках система оповіщується за допомогою механізму створення під час виконання заданої кількості об’єктів. Слід сказати, що як в тестовому режимі, так і в режимі роботи з реальними сервісами, системі достатньо однієї ініціалізації, щоб обробити будь-яку кількість змін, що задані інструкціями.

Роботу розробленої системи динамічного завантаження класів у розподілених сховищах даних в оперативній пам’яті при обробці 10 надійшовших змінених класів, в яких змінено ім’я одного з полів зображено на

рисунку 4.3.

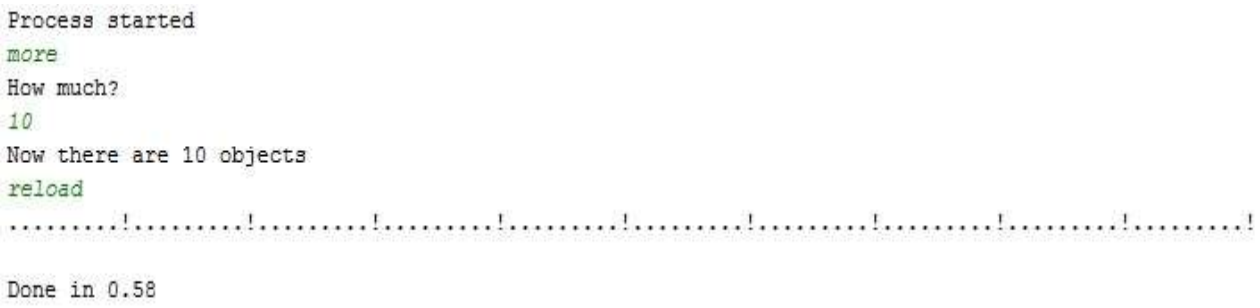

Рисунок 4.3 – Робота системи для 10 об’єктів зі зміненим іменем поля

Роботу розробленої системи динамічного завантаження класів у розподілених сховищах даних в оперативній пам’яті при обробці 100 надійшовших змінених класів, в яких змінено ім’я одного з полів зображено на рисунку 4.4.

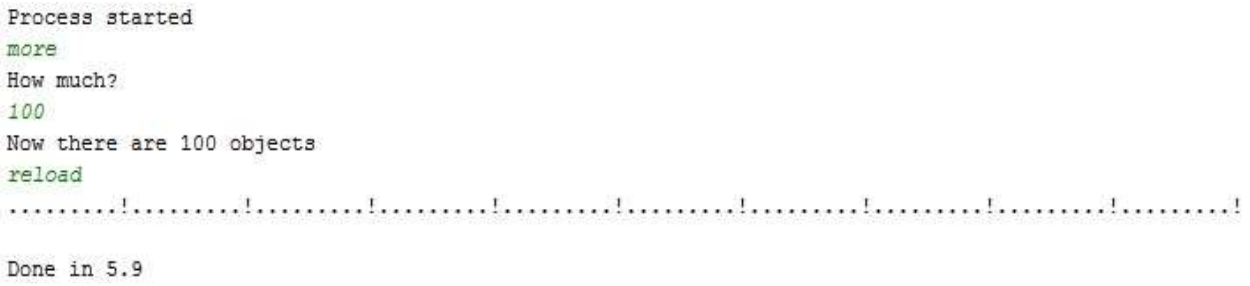

Рисунок 4.4 – Робота системи для 100 об’єктів зі зміненим іменем поля

Роботу розробленої системи динамічного завантаження класів у розподілених сховищах даних в оперативній пам’яті при обробці 1000 надійшовших змінених класів в яких змінено ім’я одного з полів зображено на рисунку 4.5.

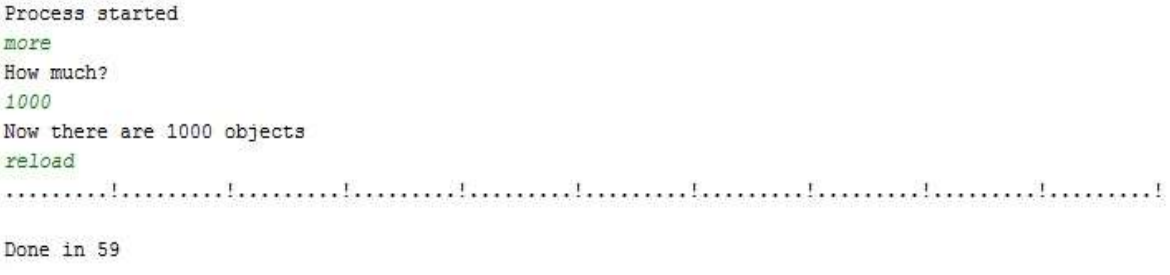

Рисунок 4.5 – Робота системи для 1000 об’єктів зі зміненим іменем поля

У відповідності до результатів, можна стверджувати, що при зміні лише імені одного з полів у зміненому об’єкті, завантаження класів відбувається лінійно, і залежить напряму від кількості об’єктів, що необхідно перезавантажити.

На перезавантаження 10 класів було витрачено лише близько 0.5 мілісекунди. Для 100 і 1000 класів 5.9 та 59 мілісекунд відповідно.

Тестування при зміні типу поля. Відповідно до другого варіанту тестування, в змінюваному об’єкті змінюється тип одного з полів, рисунок 3.6.

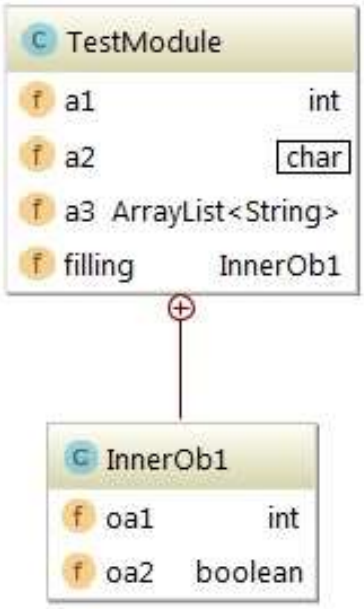

Рисунок 4.6 – Клас зі зміненим типом поля

Даний варіант тестування обумовлений тим, що зазвичай на практиці найбільші складності трапляються в ситуаціях, коли дані одного типу необхідно перетворити на зовсім інші.

Основні труднощі виникають тоді, коли саме на цьому полі зав’язана подальша логіка правильного функціонування усього сервісу загалом. Відповідно об’єкти всіх рівнів повинні бути сповіщеними про ці зміни, що б не допустити помилок у роботі, а бо ж взагалі уникнути призупинення сервісу.

Яскравим прикладом є ситуації пов’язані з перетворенням інформації з

числових типів даних у строкові.

Роботу розробленої системи динамічного завантаження класів у розподілених сховищах даних в оперативній пам’яті при обробці 10 надійшовших змінених класів, в яких змінено тип одного з полів зображено на рисунку 4.7.

```
Process started
more
How much?
10
Now there are 10 objects
reload
.........!.........!.........!.........!.........!.........!.........!.........!.........!.........!

Done in 1.4
```

Рисунок 4.7 – Робота системи для 10 об’єктів зі зміненим типом поля

Роботу розробленої системи динамічного завантаження класів у розподілених сховищах даних в оперативній пам’яті при обробці 100 надійшовших змінених класів, в яких змінено тип одного з полів зображено на рисунку 4.8.

```
Process started
more
How much?
100
Now there are 100 objects
reload
.........!.........!.........!.........!.........!.........!.........!.........!.........!.........!

Done in 14
```

Рисунок 4.8 – Робота системи для 100 об’єктів зі зміненим типом поля

Роботу розробленої системи динамічного завантаження класів у розподілених сховищах даних в оперативній пам’яті при обробці 1000 надійшовших змінених класів, в яких змінено тип одного з полів зображено на рисунку 4.9.

```
Process started
more
How much?
1000
Now there are 1000 objects
reload
.........!.........!.........!.........!.........!.........!.........!.........!.........!.........!

Done in 142
```

Рисунок 4.9 – Робота системи для 1000 об’єктів зі зміненим типом поля

В даному випадку тестування видно, що перетворення об’єктів одного типу до іншого цілком суттєво впливає на сумарний час роботи системи.

На перезавантаження 10 класів було витрачено лише близько 1.4 мілісекунди. Для 100 і 1000 класів 14 та 142 мілісекунд відповідно.

Тестування при зміні імені та типу поля у вкладеному об’єкті. В даному варіанті зміни є більш складними та суттєвими. Як і в перших двох варіантах обробка відбувається над певним полем, рисунок 4.10. Ключовою ж відмінністю є те, що це поле знаходиться не в самому змінюваному об’єкті, а у внутрішньому, і саме за рахунок рефлексивного збору залежностей даний тест найбільш чітко та якісно показує співвідношення між швидкістю та надійністю розробленої системи.

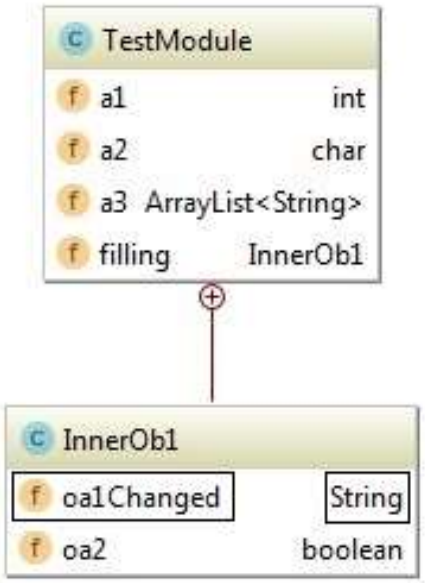

Рисунок 4.10 – Клас зі зміненим іменем і типом поля у вкладеному об’єкті

Цікавим є те, що в даній ситуації системі необхідно не лише скопіювати клас, взяти інформацію із клієнтських інструкцій, замінити необхідні значення полів та оповістити систему, але й витратити час на певні маніпуляції із внутрішнім об’єктом, саме в якому зміни і відбулися. Це може суттєво вплинути на час виконання, про те так заміна є доволі поширеною проблемою в різних сервісах, що швидко розвиваються.

Роботу розробленої системи динамічного завантаження класів у розподілених сховищах даних в оперативній пам’яті при обробці 10 надійшовших змінених класів, в яких змінено ім’я та тип одного з полів вкладеного об’єкта зображено на рисунку 4.11.

```
Process started
more
How much?
10
Now there are 10 objects
reload
.........!.........!.........!.........!.........!.........!.........!.........!.........!.........!

Done in 3.7
```

Рисунок 4.11 – Робота системи для 10 об’єктів зі зміненим ім’ям та типом поля у вкладеному об’єкті

Роботу розробленої системи динамічного завантаження класів у розподілених сховищах даних в оперативній пам’яті при обробці 100 надійшовших змінених класів, в яких змінено ім’я та тип одного з полів вкладеного об’єкта зображено на рисунку 4.12.

```
Process started
more
How much?
100
Now there are 100 objects
reload
.........!.........!.........!.........!.........!.........!.........!.........!.........!.........!

Done in 39
```

Рисунок 4.12 – Робота системи для 100 об’єктів зі зміненим ім’ям та типом поля у вкладеному об’єкті

Роботу розробленої системи динамічного завантаження класів у розподілених сховищах даних в оперативній пам’яті при обробці 1000 надійшовших змінених класів, в яких змінено ім’я та тип одного з полів вкладеного об’єкта зображено на рисунку 4.13.

```
Process started
more
How much?
1000
Now there are 1000 objects
reload
.........!.........!.........!.........!.........!.........!.........!.........!.........!.........!

Done in 401
```

Рисунок 4.13 – Робота системи для 1000 об’єктів зі зміненим ім’ям та типом поля у вкладеному об’єкті

Згідно з результатами тестів, у даному випадку, як і у попередніх варіантах, прослідковується чітка залежність часу виконання роботи системи від кількості змінених об’єктів, що надходять від клієнта.

Завдяки лінійним результатам виконання завантажень у змінених об’єктах, цілком можна стверджувати, що дана система є стабільною. Звичайно ж, для великих промислових систем випадок зі зміною внутрішнього об’єкту є одним із найбільш поширених. Відповідно, з кожним новим вкладеним класом, що необхідно перезавантажити – збільшується і загальний час виконання. При тому, саме у даному випадку цей час є доволі суттєвим, на відміну від попередніх варіантів тестування системи динамічного завантаження класів в розподілених сховищах даних в оперативній пам’яті.

Незважаючи на збільшений час роботи, система впоралася із завданням, і тому цілком може бути використана при повноцінній роботі різних сервісів.

На перезавантаження 10 класів було витрачено лише близько 1.4 мілісекунди. Для 100 і 1000 класів 14 та 142 мілісекунд відповідно.

4.2 Аналіз результатів тестування системи

Маючи тестові дані маємо можливість провести повний аналіз розробленої системи. Перед порівнянням встановлених результатів, слід зазначити загальну працездатність системи, а також перевірити механізм очищення системи від не актуальних класів. Загальний процес роботи системи, з очищенням ресурсів, зображено на рисунку 4.14.

```
"C:\Program Files\Java\jdk1.8.0_60\bin\java" ...
Process started
check
checked
more
How much?
10
Now there are 10 objects
more
How much?
100
Now there are 110 objects
more
How much?
1000
Now there are 1110 objects
clear
cleared
more
How much?
1
Now there are 1 objects
check
checked
exit

Process finished with exit code 0
```

Рисунок 4.14 – Загальний процес роботи системи

Отже, можна зробити висновок, що система функціонує саме так, як і було задумано. Усі необхідні дані, в разі потреби будуть видалені за допомогою звичайної консольної команди, що забезпечить економію ресурсів системи.

Для більш коректного аналізу отриманих даних, а також для забезпечення

мінімальної похибки, було вирішено розбити отримані результати на підгрупи, як і було зазначено під час тестування.

Спочатку порівняємо результати для всіх трьох типів завдань для 10 змінених об’єктів, рисунок 4.15.

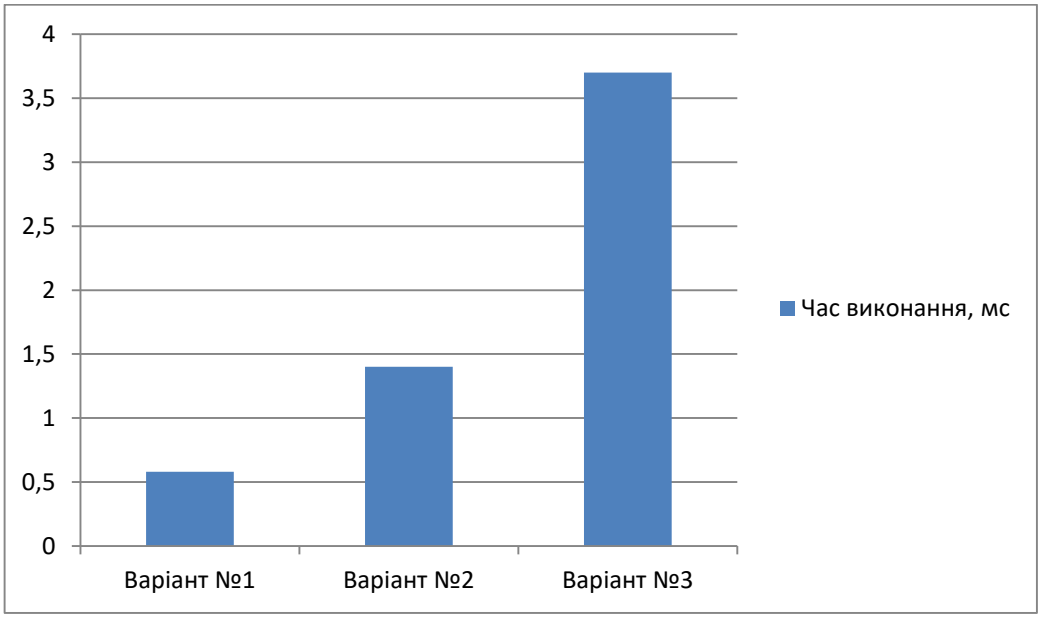

Рисунок 4.15 – Гістограма результатів тестів для 10 класів

Як можна побачити на гістограмі, при 10 об’єктах, що були передані до перезавантаження, залежність швидкості виконання перезавантаження зміненого класу прямо пропорційна складності змін у ньому.

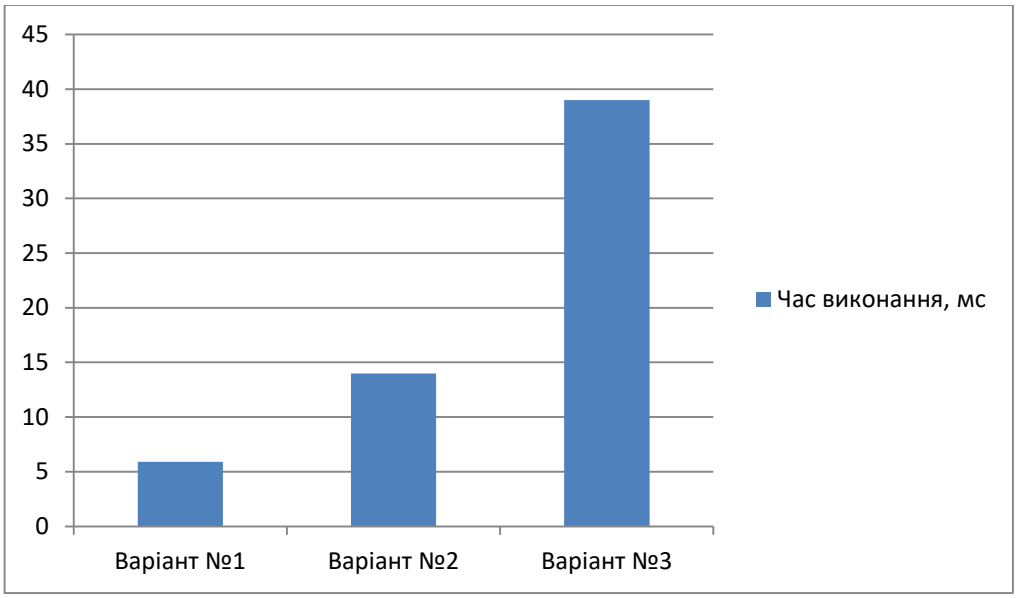

Рисунок 4.16 – Гістограма результатів тестів для 100 класів

Згідно до цієї гістограми, при 100 об’єктах, що були передані до перезавантаження, результат пропорційний до попереднього тесту.

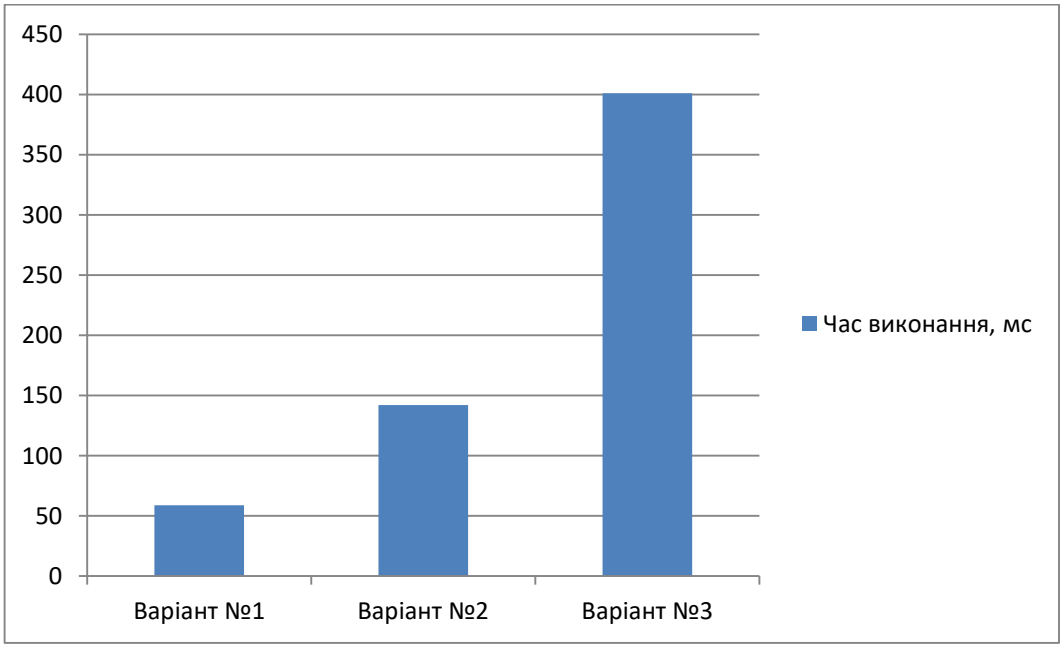

Рисунок 4.17 – Гістограма результатів тестів для 1000 класів

На усіх трьох гістограмах спостерігається лінійна залежність між кількістю складністю змін, що зазнав змінений клас, і відповідно які необхідно

перезавантажити до розподіленого сховища даних в оперативній пам’яті, та часом, за який їх було перезавантажено.

Щоб впевнитися у цьому, було вирішено провести додатковий тест, по найбільш ресурсоємному варіанту тестування, тобто за умови, коли змінюється і ім’я поля і його тип у внутрішньому класі на 100 000 об’єктах, рисунок 4.18.

```
"C:\Program Files\Java\jdk1.8.0_60\bin\java" ...
Process started
more
How much?
100000
Now there are 100000 objects
reload
.........!.........!.........!.........!.........!.........!.........!.........!.........!.........!

Done in 40403
```

Рисунок 4.18 – Робота системи для 100 000 об’єктів зі зміненим ім’ям та типом поля у вкладеному об’єкті

Висновком є однозначне підтвердження наявної залежності. Якщо порівнювати між собою усі результати тестів по трьом критеріям за однаковою кількістю об’єктів, що необхідно перезавантажити, простежується доволі чітка різниця. При першому варіанті завантаження, швидкість виконання приблизно в 2.5 рази перевищує швидкість другого варіанта завантаження, який в свою чергу майже на стільки ж перевищує по швидкості третій варіант.

Наостанок було вирішено перевірити наскільки сильно впливає кількість змінених об’єктів що мають бути перезавантажені до розподіленого сховища даних в оперативній пам’яті, рисунок 4.19.

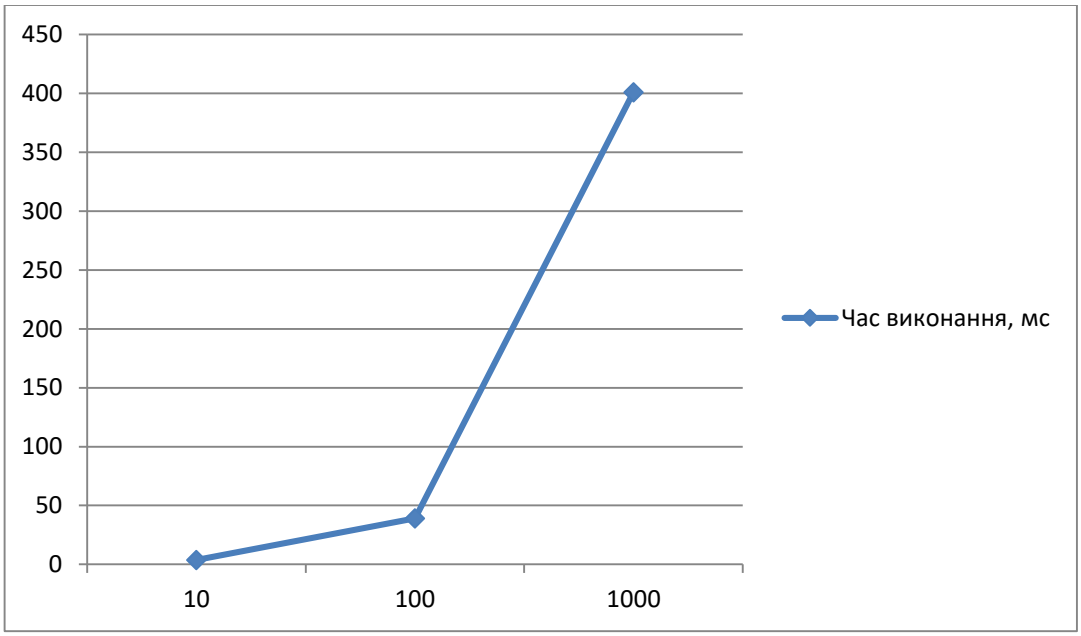

Рисунок 4.19 – Графік залежності часу від кількості перезавантажуємих об’єктів

Згідно отриманих даних, прослідковується прямопропорційна залежність між часом виконання операцій, та кількості об’єктів, над якими вони виконуються. При збільшенні кількості об’єктів для завантаження приблизно в 10 разів, час за який вони будуть завантажені також збільшується на стільки ж, з не вагомою похибкою, про що свідчать дані що були здобуті як для 10 об’єктів, так і для 100 000 об’єктів – 3.7 мілісекунд та 40 403 мілісекунд відповідно.

Отже, однозначним висновком є те, що система працює стабільно і цілком відповідає критерію надійності на протестованих наборах даних.

Підбиваючи підсумки, слід зазначити, що розроблена система можлива у використанні для будь-яких сучасних сервісів, що використовують IMDG.

Найбільш ефективним буде застосування системи динамічного завантаження класів в розподілених сховищах даних в оперативній пам’яті для більш простих сервісів, що не мають великої кількості внутрішніх об’єктів. Саме рефлексія суттєво сповільнює перезавантаження об’єкту. У випадках, коли необхідні лише поверхневі зміни основного класу – користь цієї системи суттєво

відчувається за рахунок швидкої заміни та надійності. В будь-якому разі, навіть при роботі із великою кількістю внутрішніх класів, на практиці набагато вигіднішим є перезавантажити зміни в процесі роботи сервісу, аніж витрачати час на перезавантаження усього сховища, або ж окремої частини ресурсу.

ВИСНОВКИ ДО РОЗДІЛУ 4

В даному розділі була протестована система динамічного завантаження класів у трьох варіантах роботи. Першим варіантом було обрано ситуацію, коли у зміненому класі змінилося лише ім’я поля. Другим варіантом стала ситуація, коли імена полів залишалися незмінними, про те змінювався тип одного з них. Найскладнішим же варіантом став третій – варіант, під час якого відбувалися зміни як імені поля у класі, так і його типу, складнощів додавав той факт, що даний об’єкт був внутрішнім об’єктом іншого класу.

Аналіз результатів тестування показав, що робота системи прямо пропорційно залежить від двох основних факторів: складності зміни у об’єкті та кількості цих самих об’єктів.

Було встановлено, що система працює з різною кількості даних, при цьому залишаючись стабільною, що свідчить про надійність розробленої системи динамічного завантаження класів у розподілених сховищах даних в оперативній пам’яті.

Таймінг роботи розробленої мною системи показав, що при використанні великої кількості внутрішніх класів, що втрати часу на перезавантаження змін в процесі роботи сервісу значно нижчі, аніж витрата часу на перезавантаження усього сховища, або ж окремої частини ресурсу.

ВИСНОВКИ

У відповідності до поставлених в дослідженні задач, мною були зроблені висновки, наведені нижче.

1. Аналіз технічної та наукової літератури із тематики дослідження дозволив виокремити основні теоретичні положення щодо розробки та використання розподілених не реляційних баз даних в оперативній пам’яті – IMDG. Були визначені причини виникнення такої технології, її недоліки та актуальні проблеми. Аналіз наявних інструментів для динамічного завантаження класів виявив протиріччя між потребою у використанні таких класів та відсутністю вже готових рішень, що підтвердило актуальність дослідження. Також були вивчені існуючи сервіси, що надають доступ до баз даних з архітектурою in-memory-data-greed.
2. . В межах дослідження було змодельовано алгоритм роботи інструменту динамічного завантаження класів у розподілених сховищах даних та визначена його ефективність, а саме, було показано, що запропонований підхід динамічного завантаження дозволяє уникнути перезавантаження системи, або окремої частини, і, відповідно, при необхідності внесення певних змін, повністю запобігає втраті даних, і не потребує часу на перезапуск.
3. На базі запропонованої моделі був розроблений алгоритм роботи інструменту динамічного завантаження класів без втрати даних та розглянуті особливості його застосування в умовах розподіленості та описана реалізація розробленого алгоритму. Описані головні класи програми та їх функції. Також було запропоновано додатковий спосіб прискорення процесу завантаження у разі нестачі доступної для цього пам’яті.

4. Спираючись на результати теоретичного дослідження, було створено програмний модуль реалізації інструментів динамічного завантаження класів у розподілених сховищах даних з архітектурою IMDG та протестовано розроблену систему динамічного завантаження класів у розподілені сховища даних в оперативній пам’яті. Порівняний результат між різними варіантами тестування. Проаналізована залежність швидкості роботи системи від таких чинників, як кількість об’єктів, що необхідно перезавантажити, а також від складності змін у цих об’єктах. Продемонстровано надійність розробленої системи.

Таким чином, всі поставлені задачі в дослідженні були виконані у повному обсязі.

ЛІТЕРАТУРА

1. Бузовский О.В., Подрубайло А.А. Актуальные проблемы распределенных хранилищ данных в оперативной памяти. // Вісник Національного авіаційного університету. Проблеми інформатизації та управління. – 2015 – №2(50) 2015 – С. 36-43
2. Подрубайло А.А. Повышение производительности хранилищ данных в оперативной памяти посредством использования программной транзакционной памяти [Електронний ресурс]. Режим доступу: http://cyberleninka.ru/article/n/povyshenie-proizvoditelnosti-hranilisch-dannyh-v-operativnoy-pamyati-posredstvom-ispolzovaniya-programmnoy-tranzaktsionnoy-pamyati (дата звернення: 09.10.2017)
3. Бузовский О.В., Подрубайло А.А. Методы и алгоритмы обьеденинения таблиц для распределенных хранилищ данных в оперативной памяти [Електронний ресурс]. Режим доступу: http://it-visnyk.kpi.ua/wp-content/uploads/2015/03/issue-60_p74-83.pdf (дата звернення: 09.10.2017)
4. Roy Prins. In-Memory Data Grids [Електронний ресурс]. Режим доступу: https://dzone.com/articles/memory-data-grids?ref=dzone (дата звернення: 16.10.2017)
5. Что такое In-Memory Data Grid [Електронний ресурс]. Режим доступу: https://habrahabr.ru/post/160517/ (дата звернення: 16.10.2017)
6. In-memory-data-grid. Масштабируемые хранилища данных [Електронний ресурс]. Режим доступу: https://habrahabr.ru/post/126580/ (дата звернення: 16.10.2017)
7. Internals of Java Class Loading [Електронний ресурс]. Режим доступу: http://www.onjava.com/pub/a/onjava/2005/01/26/classloading.html (дата звернення: 17.10.2017)
8. Pivotal. Apache Geode 1.0.0-incubating Documentation [Електронний ресурс]. Режим доступу: http://geode.docs.pivotal.io/ (дата звернення: 17.10.2017)

9. Pivotal. Topology Types [Електронний ресурс]. Режим доступу: http://geode.docs.pivotal.io/docs/topologies_and_comm/topology_concepts/topology_types.html (дата звернення: 17.10.2017)
10. Pivotal. How Member Discovery Works [Електронний ресурс]. Режим доступу: http://geode.docs.pivotal.io/docs/topologies_and_comm/topology_concepts/how_member_discovery_works.html (дата звернення: 18.10.2017)
11. VFabric 5 Documentation Center [Електронний ресурс]. Режим доступу: https://pubs.vmware.com/vfabric5/index.jsp?topic=/com.vmware.vfabric.gemfire.6.6/deploying/gfsh/gfsh_cmd_line_options.html (дата звернення: 18.10.2017)
12. Jar Class Loader [Електронний ресурс]. Режим доступу: https://apache.googlesource.com/incubator-geode/+/sga2/gemfire-core/src/main/java/com/gemstone/gemfire/internal/JarClassLoader.java (дата звернення: 20.10.2017)
13. Bill Bejeck. Using Java 7's WatchService to Monitor Directories [Електронний ресурс]. Режим доступу: https://dzone.com/articles/using-java-7s-watchservice (дата звернення: 20.10.2017)
14. Oracle. Watching a Directory for Changes [Електронний ресурс]. Режим доступу: https://docs.oracle.com/javase/tutorial/essential/io/notification.html (дата звернення: 21.10.2017)
15. IBM Bluemix. Теория и практика Java: Динамическая компиляция и измерение производительности [Електронний ресурс]. Режим доступу: http://www.ibm.com/developerworks/ru/library/j-jtp12214/ (дата звернення: 22.10.2017)
16. Динамическая компиляция Java кода [Електронний ресурс]. Режим доступу: https://habrahabr.ru/company/haulmont/blog/248981/ (дата звернення: 22.10.2017)